Condicionales y bucles en Python (con NumPy)

Uso de NumPy

En este tutorial vamos a usar **NumPy** para todas las operaciones matemáticas y para trabajar con vectores y matrices.

- NumPy es una librería de Python para cálculo numérico eficiente.
- Siempre que usemos vectores o matrices, los representaremos como **arrays de NumPy**, no como listas de Python.
- Para usar NumPy, al principio de cada programa hay que escribir:

```
import numpy as np
```

En los ejemplos que siguen supondremos que esa línea aparece al comienzo de nuestros ficheros de Python.

Instrucción `if`

La instrucción `if` nos permite preguntar una condición respecto a los cálculos que estamos manejando y obrar en consecuencia en función de si dicha condición se cumple o no. La forma básica de esta instrucción en Python es la siguiente:

```
if condicion:
    instrucciones
```

Si la condición se cumple (es decir, si es `True`), entonces las instrucciones se ejecutarán. Si no se cumple (es `False`), las instrucciones se ignorarán.

Muy importante sobre la identación en Python. A diferencia de lenguajes como Octave o C, en Python *no* se usan palabras como `end` para marcar el final de un bloque. En su lugar:

- El carácter `:` al final de la línea (por ejemplo en `if condicion:`) indica el comienzo de un bloque.
- *Todas* las líneas que pertenecen a ese bloque tienen que ir *indentadas* (con espacios en blanco al principio de la línea) con la misma profundidad.
- Cuando la identación vuelve hacia atrás (disminuye), el bloque termina.

Es decir, **la identación (los espacios al principio de la línea) es la manera en que Python sabe dónde empieza y termina un bloque de código**. Si la identación es incorrecta, el programa dará un error de sintaxis o se comportará de forma inesperada.

Veamos un ejemplo sencillo de cómo implementaríamos la función del valor absoluto $|x|$ si ésta no apareciera implementada en Python (aunque en realidad ya existe la función `abs`):

```
import numpy as np

def valor_absoluto(x):
    if x < 0:
        y = -x
    if x >= 0:
        y = x
    return y
```

Los operadores de comparación más usados con estas instrucciones son los que se muestran a continuación:

Operador	Interpretación
$<$	menor que
$<=$	menor que o igual a
$>$	mayor que
$>=$	mayor que o igual a
$==$	igual a
$!=$	distinto de

En Python se pueden agrupar varias condiciones para una misma instrucción `if` y hacer que la instrucción se ejecute cuando se cumplan todas, o se cumpla alguna. En tal caso, habrá que poner cada condición entre paréntesis y separarlas por el operador lógico correspondiente. Los conectores lógicos que usa Python aparecen en la siguiente tabla:

Símbolo	Operador lógico en Python
`not`	no
`and`	y
`or`	o

Pongamos por ejemplo que queremos calcular las raíces de una parábola y que queremos que el cálculo se produzca solamente cuando tengamos la certeza de que las raíces existen y son reales. Como sabemos que las raíces de la parábola $ax^2 + bx + c$ vienen dadas por

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},$$

podemos programar el código en Python (usando NumPy) como sigue:

```
import numpy as np

if (b**2 - 4*a*c >= 0) and (a != 0):
    x1 = (-b + np.sqrt(b**2 - 4*a*c)) / (2*a)
    x2 = (-b - np.sqrt(b**2 - 4*a*c)) / (2*a)
```

De nuevo, fíjate en que todas las instrucciones que pertenecen al bloque del `if` (los cálculos de `x1` y `x2`) están indentadas con los mismos espacios.

Instrucciones `if` anidadas

El comando `else` nos permitirá introducir una instrucción en el caso de que la condición dada por el comando `if` correspondiente no se cumpla. Así, por ejemplo, la función para el valor absoluto podría implementarse de la siguiente forma en Python:

```
import numpy as np

def valor_absoluto(x):
    if x < 0:
        y = -x
    else:
        y = x
    return y
```

Este procedimiento funciona si sólo tenemos que considerar dos situaciones: que una condición se cumpla o que no. En caso de que haya más de dos condiciones que determinen qué instrucciones ejecutar, deberemos usar el comando `elif` (abreviatura de "else if"). El siguiente método computa todas las soluciones reales posibles de una ecuación de segundo grado $ax^2 + bx + c = 0$:

```
import numpy as np

def ecuacion_cuadratica(a, b, c):
    if a != 0:
        # Ecuación de segundo grado
        if b**2 - 4*a*c >= 0:
            x1 = (-b + np.sqrt(b**2 - 4*a*c)) / (2*a)
            x2 = (-b - np.sqrt(b**2 - 4*a*c)) / (2*a)
            print("La ecuación tiene dos soluciones reales, que son")
            print(x1)
            print("y")
```

```
        print(x2)
    else:
        print("La ecuación no tiene soluciones reales.")
elif b != 0:
    # Ecuación lineal: b x + c = 0
    print("La ecuación es una ecuación lineal con solución")
    print(-c / b)
elif c != 0:
    print("La ecuación es una identidad falsa que, por lo tanto, no tiene solución.")
else:
    print("La ecuación es una identidad verdadera y, por lo tanto, cualquier número es
```

Observa de nuevo cómo los bloques se estructuran únicamente mediante la identación: las líneas dentro de cada `if`, `elif` o `else` están todas igualmente indentadas.

Bucles

Un bucle es una estructura de programación que permite la repetición controlada de un conjunto de instrucciones. Los bucles que veremos a continuación son los más utilizados, no sólo en Python, sino en la inmensa mayor parte de los lenguajes de programación.

- **Bucle `for`.**

  Este bucle ejecuta una determinada instrucción tantas veces como se lo indique el usuario. Su estructura en Python es la siguiente:

  ```
  for indice in iterable:
      instrucciones
  ```

  El objeto `iterable` suele ser una lista, una tupla, un rango de números, etc. En este curso, cuando trabajemos con colecciones numéricas, usaremos **arrays de NumPy**. El índice (por ejemplo, `indice`) irá tomando, en orden, los valores del iterable, y para cada uno de ellos se ejecutarán las instrucciones indicadas por el usuario.

  En particular, no hace falta que se llame `i`: cualquier otro nombre funcionará igual.

  Un caso muy importante es el uso de la función `range`, que genera secuencias de enteros. Por ejemplo, `range(1, 11)` genera los enteros del 1 al 10 (el extremo superior no se incluye).

  Si queremos realizar un código que construya el vector

  $$v = \left(1, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{10}\right),$$

  operaríamos de la siguiente manera (usando un array de NumPy):

```
import numpy as np

v = np.zeros(10)              # Creamos un vector de 10 ceros

for i in range(10):           # i toma los valores 0, 1, ..., 9
    v[i] = 1 / (i + 1)
```

Así, el vector `v` se iría creando en cada iteración del bucle `for`.

Hagamos otro programa, que proporcione la suma de los elementos de un vector de NumPy (lo cual ya podemos hacer con la función `np.sum`, pero lo implementamos a mano):

```
import numpy as np

def mi_sum(v):
    x = 0
    for i in range(v.shape[0]):
        x = x + v[i]
    return x
```

Notemos que hemos tenido que iniciar la suma con `x = 0` para poder tener la variable creada, o el programa daría problemas. Esta técnica es muy socorrida para casos donde tengamos que realizar sumas o productos con elementos cuyo límite no podemos controlar (vectores de cualquier longitud, matrices de cualquier dimensión...).

Como segundo ejemplo, y nuevamente aunque Python ya tiene una función para ello, vamos a implementar una función que nos calcule el factorial de un número:

```
import numpy as np

def factorial(n):
    # Comprobamos que n es un entero no negativo
    if (int(n) != n) or (n != abs(n)):
        print("Necesito un entero no negativo.")
        return None
    n = int(n)

    if n == 0:
        x = 1
    else:
        x = 1
```

```
        for i in range(1, n + 1):
            x = x * i
    return x
```

Obsérvese de nuevo que los bloques de código se delimitan por identación: las instrucciones dentro del `if`, `elif` o `for` están indentadas respecto a la cabecera.

Veamos otro programa que tome una matriz cuadrada y lo que haga sea reflejar la triangular superior sobre la inferior, es decir, cree una matriz simétrica a partir de los elementos de la triangular superior. Usaremos arrays de NumPy para representar matrices:

```
import numpy as np

def triang_sim(A):
    # A es una matriz (array 2D de NumPy)
    n, m = A.shape

    if n != m:
        print("La matriz introducida no es cuadrada.")
        return None
    else:
        # Copiamos A en B para no modificar la original
        B = A.copy()

        for i in range(1, n):
            for j in range(i):
                B[i, j] = A[j, i]

        return B
```

Por último, hagamos un programa que calcule el término n-ésimo de la sucesión de Fibonacci (que, como todo el mundo sabe, es una sucesión que se inicia con 0 y 1 y después calcula el siguiente término mediante la suma de los dos anteriores: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$). Aquí no necesitamos NumPy, pero mantenemos la misma sintaxis de bloques:

```
def fib(n):
    # Comprobamos que n es un entero positivo
    if (int(n) != n) or (n != abs(n)):
        print("Necesito que el número sea un entero positivo.")
        return None
    n = int(n)
```

```
    if n == 1:
        x = 0
    elif n == 2:
        x = 1
    else:
        x1 = 0
        x2 = 1
        for i in range(3, n + 1):
            x = x1 + x2
            x1 = x2
            x2 = x
    return x
```

- **Bucle `while`.**

  Los bucles `while` ejecutan unas instrucciones mientras que determinada condición lógica se verifica. La sintaxis es como sigue:

  ```
  while condicion:
      instrucciones
  ```

  De nuevo, la importancia de la identación es crucial: todas las instrucciones que se repiten mientras la condición sea verdadera van indentadas bajo la línea del `while`.

  El riesgo que pueden tener estos bucles es que la condición no deje de verificarse, y el programa se vea atrapado en un bucle infinito, en cuyo caso deberemos usar la instrucción `break` para salir de éste o, en su defecto, detener el programa.

  Nótese que un bucle `for` puede transformarse en un bucle `while`. Pongamos el ejemplo de la suma de las coordenadas de un vector de NumPy. Si queremos usar un bucle `while` podríamos escribir:

  ```
  import numpy as np

  def mi_sum_while(v):
      x = 0
      i = 0
      n = v.shape[0]

      while i < n:
          x = x + v[i]
  ```

```
            i = i + 1

        return x
```

Método de Jacobi para aproximación de soluciones de ecuaciones lineales

Con lo visto anteriormente tenemos las herramientas para hacer programas complicados, entre los que se encuentran los métodos iterativos, es decir, algoritmos que consisten en la repetición de un mismo proceso varias veces, de tal forma que se construye una sucesión. Nos vamos a detener en el llamado método de Jacobi, el cual permite encontrar soluciones de sistemas del tipo $Ax = b$ (recordemos que el cálculo de $A^{-1}b$ puede empezar a dar problemas al poco que la matriz sea complicada). El método es el que sigue:

1. Empezamos con un valor cualquiera x_0 (vector de NumPy).
2. x_n lo obtenemos del término anterior, x_{n-1}, de la siguiente forma: la coordenada i-ésima la obtenemos a partir de la fórmula

$$x_n(i) = \frac{1}{a_{ii}} \left(-\sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_{n-1}(j) + b_i \right), \quad i = 1, \ldots, n.$$

Si consideramos ese método, la sucesión $\{x_n\}$ creada verifica que tiende a la solución del sistema $Ax = b$ (bajo ciertas condiciones sobre la matriz A). En particular, se verifica que la distancia entre dos elementos consecutivos tiende a cero:

$$\lim_{n \to \infty} \|x_n - x_{n+1}\| = 0.$$

Así, si queremos implementar el método con 100000 iteraciones, podríamos escribir el siguiente código (suponiendo que A es un array 2D de NumPy, y x y b son arrays 1D de NumPy):

```
import numpy as np

# Suponemos que A, b y x están definidos, y que A es cuadrada.

for k in range(100000):
    y = x.copy()                # Copia de la aproximación anterior
    n = x.shape[0]

    for i in range(n):
```

```
        cuant = 0
        for j in range(n):
            cuant = cuant + A[i, j] * y[j]

        x[i] = (-cuant + A[i, i] * y[i] + b[i]) / A[i, i]
```

De nuevo, fíjate en la estructura de identación: el bloque del bucle externo `for k in range(100000):` lleva indentado todo el código que se repite en cada iteración, y dentro de él, el bloque del bucle interior `for i in range(n):` va indentado un nivel más.

Ejercicio.
Adaptar el código anterior para un programa cuyas variables sean el número de iteraciones N, la matriz del sistema A, el vector de términos independientes b y el valor inicial x (todos ellos arrays de NumPy). Incluir además un parámetro de control d, de forma que si $\|x_n - x_{n-1}\| < d$ entonces el programa interrumpa el proceso (porque consideraremos que estamos lo suficientemente cerca de la solución).

Es decir, el programa debería tener como cabecera algo parecido a

```
import numpy as np

def jacobi(N, A, b, x, d):
```

y en su interior, dentro de un bucle, ir actualizando la aproximación `x`. En cada paso se deberá calcular la norma de la diferencia entre la nueva aproximación y la anterior usando, por ejemplo, la norma euclídea que proporciona NumPy:

$$\|x_n - x_{n-1}\| = \sqrt{\sum_{i=1}^{n} (x_n(i) - x_{n-1}(i))^2}$$

lo cual en Python con NumPy se puede escribir como

```
np.linalg.norm(x_n - x_anterior)
```

y, si es menor que `d`, salir del bucle usando la instrucción `break`. Recuerda usar la identación correctamente en todos los bloques (`for`, `if`, `while`, etc.) para que el programa funcione.