Taller de Física Computacional

Iteración sobre secuencias

Cristián G. Sánchez y Carlos J. Ruestes

2021

Iteración sobre secuencias

Iteración sobre secuencias

Python provee la declaración `for` para iterar sobre secuencias:

```
for x in s:
    # esto se repite tantas veces como elementos
    # tenga s y x toma el valor i-esimo en la iteración i
```

Dada una secuencia, por ejemplo una lista `s = [1,2,3,4,5]`, el bloque que sigue a la línea contiendo la declaración `for` se ejecutará 5 veces. En cada iteración la variable `x` tomará los valores 1,2,3,4 y 5.

Iteración sobre un rango

Iteración sobre un rango

Los rangos se utilizan en muchos casos para representar una secuencia de enteros sobre la cual iterar sin necesariamente construir una lista que contenga los valores:

```
for i in range(0,n):
    # esto se repite n veces, i toma los valores desde
    # 0 a (n-1)
```

Esta es, por lejos, la estructura de bucle más utilizada en Python y representa una alternativa a la declaración `while`.

Las palabras clave `break` y `continue` se pueden utilizar de la misma manera que en el bucle `while`. `break` sale incondicionalmente y `continue` hace que i tome el siguente valor y el bloque se ejecute desde el inicio salteando lo que siga.

Iteración sobre un `enumerate`

Iteración sobre un `enumerate`

La declaración `for` se puede utilizar con objetos *iterables* o funciones especiales llamadas *generadores*. Si bien no vamos a entrar en esa rama hay un generador que es muy útil: `enumerate(s)` devuelve, en cada iteración, una tupla `(i,x)` que contiene el valor y su índice:

```
for i,x in enumerate(s):
    # esto se repite tantas veces como elementos
    # tenga s y x e i toman el valor i-ésimo y su índice,
    # respectivamente, en cada iteración.
```

Ejemplo

Evaluar una función en un intervalo de la recta real

Podemos modificar el ejemplo que habíamos utilizado en la demo de bucles para, ahora que sabemos como, almacenar una tabla de valores en vez de mostrarla en pantalla:

```
n = 1000; xmin = -1.0; xmax = 1.0
xs = []; ys =[]

for i in range(0,n+1):
    x = xmin + i * (xmax - xmin) / n
    xs += [x]
    ys += [f(x)]
```

Ahora la lista `xs` contiene la grilla sobre la que evaluamos a la función y la lista `ys` el valor en cada punto. Notar que usamos `(n+1)` en el rango, `n` es la cantidad de sub-intervalos en la grilla que contiene `(n+1)` puntos.

Comprehensiones

Comprehensiones

Las *comprehensiones* proveen una forma concisa de crear listas. Las comprhensiones utilizan una o más palabras clave `for` o `if`

```
# lista de numeros entre 0 y 99
s = [i for i in range(0,100)]

# números pares entre 0 y 99
s = [i for i in range(0,100) if i%2 ==0]

# pueden anidarse
s = [[i*j for i in range(0,100)] for j in range(0,100)]
```

En estos ejemplos hemos usado rangos pero se puede utilizar cualquier iterable.

Mapas

Mapas

Es posible construir un generador que represente la aplicación de una función a cada miembro de una lista utilizando la función `map()`

```
# lista de numeros entre 0 y 99
xs = [-1 + 2 * i / 100 for i in range(0,101)]

# una función
def f(x):
    return np.sin(x)/(x+3)

# lista de valores de f sobre xs
ys = list(map(f,xs))
```

En el último ejemplo convertimos el iterable en una lista.

Funciones anónimas

Funciones anónimas, expresiones `lambda`

Para evitar definir la función en el ejemplo anterior podemos utilizar la expresión `lambda` que nos permite definir una función *anónima* en el lugar en el que la necesitamos.

```
# lista de numeros entre 0 y 99
xs = [-1 + 2 * i / 100 for i in range(0,101)]
# lista de valores de f sobre xs
ys = list(map(lambda x : np.sin(x)/(x+3),xs))
```

Las expresiones `lambda` pueden usarse en cualquier lugar que debamos pasar una función como argumento y permiten usar la función sin definirla explícitamente.

Síntesis y recursos:

- Manual de referencia de Python
- Manual de la Librería estándar de Python