Música por computadora

Ernesto Romero y Hernani Villaseñor
Centro Multimedia 2012

Sesión 3

3.3 Funciones, Arreglos

Funciones

Las funciones en SC se encasillan entre llaves { }. Una función en SC representa una acción, por ejemplo hacer sonar un sonido o ejecutar una rutina. Las funciones por sí mismas no trabajan, necesitan de un mensaje para saber que hacer.

La siguiente función es una onda sinoidal, que mediante el mensaje .play sonará cuando evaluemos la línea de código. (recuerden encender el servidor con s.boot)

```
{SinOsc.ar (440, 0, 0.5)}.play
```

Las funciones también trabajan del mismo modo que en la forma tradicional matemática f(x). Por ejemplo si queremos hacer la función $f(x)=x^2$ podemos escribirla así:

```
f={|x| x**2}
```

Los caracteres | | indican que x es el argumento que recibirá el valor al que queramos aplicar la función.

Para dar un valor a x lo hacemos con el método o mensaje .value() . La siguiente línea da el valor 2 a x pidiendo así 2^2

```
f.value(2)
```

Al evaluar la función obtenemos, por supuesto, el número 4 en la post window.

Podemos usar mas de un argumento en la función. Por ejemplo para una suma de cuadrados:

```
f={|a, b| (a**2)+(b**2)}
f.value(3,4)
```

El resultado debe ser 25
O para la hipotenusa de un triángulo rectángulo. Recordemos el teorema de Pitágoras

$a^2 + b^2 = c^2$

```
f={|a, b| ((a**2)+(b**2)).sqrt}
f.value(3,4)    //  el resultado es 5
```

Se puede usar una función dentro de otra operación:

```
10*f.value(3,4)   //  el resultado es 50
```

Una función solo arroja el resultado de lo último que tenga escrito dentro de ella. En el siguiente ejemplo solo obtendremos el resultado de la suma a + b y no el de la multiplicación a * b

```
f={|a, b| a*b; a+b}
f.value(2,3)    // el resultado es 5
```

Una misma función puede ejecutarse cualquier cantidad de veces con el mensaje .do

```
5.do{"hola mundo".postln}
```

El resultado en la post es:

hola mundo
hola mundo
hola mundo
hola mundo
hola mundo
5

Para el siguiente ejemplo utilizaremos el mensaje .rrand
Si enviamos el mensaje .rrand a un número *a* y agregamos un número *b* como argumento podemos obtener un valor aleatorio dentro del rango entre *a* y *b*.

```
a.rrand(b)
```

Así si *a*=300 y *b*=500 obtendremos un número aleatorio entre 300 y 500

```
300.rrand(500)
```

Con este método .rrand podemos, por ejemplo, producir varias ondas senoidales.

```
10.do{{SinOsc.ar(300.rrand(500), 0, 0.1)}.play}
```

Notar que hay una función dentro de otra y por eso usamos llaves anidadas {{}}. También estamos escribiendo 0.1 en el argumento amp del SinOsc para bajar el volumen de cada onda, de esta forma al sumarse el volumen de cada una de las 10 ondas tenemos un volumen total de 1

SuperCollider tiene definida la función de iteración de la suma automáticamente. Una iteración es una operación que se realiza sobre el resultado de la misma operación las veces que se desee. Si partimos del 0, y la operación es sumar 1 al resultado, obtenemos una secuencia de números enteros:

```
0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 1 = 4
```

Para ver esto simplemente hay que declarar la siguiente línea:

10.do{ | i | i.postln }

En la post se imprime:

0
1
2
3
4
5
6
7
8
9
10

El 10 no fué un valor de i. El 10 se imprime por que la función se realizó 10 veces.

Lo que está sucediendo aqui es que el argumento i esta siendo iterado con la operación + 1

Con esta iteración, y el mensaje .do , podemos hacer funciones como la siguiente, donde al número 40 se le va sumando la iteración del argumento i dando como resultado los números del 40 al 49 (del 40 al 49 hay 10 números)

```
10.do{ | i | (40 + i).postln}
```

En la post se imprime:

40
41
42
43
44
45
46
47
48
49
10

El 10 no fué un valor de (40 + i). El 10 se imprime por que la función se realizó 10 veces.

Más aún, si usamos la suma 40 + i como argumento para una onda sinusoidal SinOsc podemos escuchar lo siguiente:

```
10.do{ | i | {Pulse.ar( 40 + i )}.play}
```

Arreglos

Los arreglos o *arrays* son parte del lenguaje de programación en general. Los arreglos son conjuntos ordenados. En SC los arreglos se encasillan entre corchetes [ ]. Sirven para contener objetos o valores de los cuales podemos hacer uso de diversas formas.

Ejemplo de un arreglo en SC con números

```
[1, 2, 3, 4, 5]
```

Ejemplo de un arreglo con números y *strings*

```
[1, 'nada', 3, 'algo']
```

Existen gran cantidad de mensajes que se pueden enviar a un arreglo. A continuación enlistamos algunos:

```
[0, 1, 2, 3, 4].reverse   //  pone el arreglo en reversa:  [4,
3, 2, 1, 0]
```

```
[0, 1, 2, 3, 4].scramble  //  desordena aleatoriamente el
arreglo: [2, 1, 4, 0, 3]

[0, 1, 2, 3, 4].plot  //  crea una gráfica en 2D

[0, 1, 2, 3, 4].scramble.plot // Se pueden aplicar mensajes
consecutivos

[0, 1, 2, 3, 4].mirror //  crea un espejo del arreglo: [0, 1, 2,
3, 4, 3, 2, 1, 0]

[0, 1, 2, 3, 4].sum //  suma los elementos del arreglo :
0+1+2+3+4=10

[0, 1, 2, 3, 4].pyramid // crea una estructura piramidal con los
elementos del arreglo: [ 0, 0, 1, 0, 1, 2, 0, 1, 2, 3, 0, 1, 2,
3, 4 ]
```

La forma piramidal se percibe más facilmente si la reescribimos así:

```
[ 0,
  0, 1,
  0, 1, 2,
  0, 1, 2, 3,
  0, 1, 2, 3, 4 ]
```

También podemos aplicar operaciones a arreglos:

```
[0, 1, 2, 3, 4] + 10  //  Suma 10 a cada elemento:  [10, 11, 12,
13, 14]
[0, 1, 2, 3, 4] * 10   //  Multiplica por 10 cada elemento:  [0,
10, 20, 30, 40]
```

Se pueden hacer también operaciones entre arreglos. La correspondencia en estos casos es de uno a uno. Si se tienen dos arreglos *a* y *b* y se aplica una suma entre ellos el primer elemento de *a* se sumará al primer elemento de *b*, el segundo con el segundo y así sucesivamente.

```
[0, 1, 2, 3, 4] + [5, 6, 7, 8, 9]    //  el resultado es [5, 7,
9, 11, 13]
```

Si los arreglos tienen diferente cantidad de elementos la operación se aplicará uno a uno también, pero cuando el arreglo más corto se agote comenzará desde el principio de nuevo hasta que los elementos del arreglo más largo se terminen. En el siguiente ejemplo el arreglo *a* tiene 7 elementos mientras que el arreglo *b* tiene solo 5

```
a=[0, 1, 2, 3, 4, 5, 6]
b=[5, 6, 7, 8, 9]
a + b //  el resultado es [5, 7, 9, 11, 13, 10, 12]
```

Las sumas se realizan así:

```
[0+5, 1+6, 2+7, 3+8, 4+9, 5+5, 6+6]
```

Un atajo para crear un arreglo de números consecutivos es así:

```
(1..10)  //  crea el array [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Array es el objeto de SC para hacer arreglos con estructuras específicas sin tener que escribir cada uno de sus elementos. Existen una gran cantidad de mensajes para crear diferentes arreglos con el objeto Array. Daremos solamente tres ejemplos de mensajes, pero recomendamos estudiar todos los que vienen en el archivo de ayuda de Array y de los objetos de quien hereda.

Array.series(tamaño, comienzo, paso)

Crea una serie arímetica de valores, se definen tres argumentos que son: tamaño, comienzo y paso. Tamaño indica la cantidad de objetos o valores que contendrá el arreglo, comienzo nos dice a partir de que valor iniciar y paso es el la cantidad de valores intermedios entre cada objeto. (e.g. al valor se le suma el paso para obtener el siguiente valor)

Ejemplo de un Array usando el método .series

```
// se crea un Array de 5 valores que comienza en 2 y cuyo valor
subsecuente ira en aumento por valores de 4
Array.series(5, 2, 4)   //  el resultado es [2, 6, 10, 14, 18]
```

Array.geom(tamaño, comienzo, crecimiento)

Crea una serie geométrica de valores, se definen tres argumentos que son: tamaño, comienzo y crecimiento. Tamaño indica la cantidad de objetos o valores que contendrá el arreglo, comienzo nos dice a partir de que valor iniciar y crecimiento es el factor de crecimiento entre cada objeto. (e.g. el valor se multiplica por el factor de crecimiento para obtener el siguiente valor)

Ejemplo de un Array usando el método .geom

```
// se crea un Array de 5 valores que comienza en 2 y cuyo factor
de crecimiento es de 4
Array.geom(5, 2, 4)   //  el resultado es [2, 8, 32, 128, 512]
```

Array.rand(tamaño, mínimo, máximo)

Crea un arreglo de valores aleatorios, se definen tres argumentos que son: tamaño, mínimo y máximo. Tamaño indica la cantidad de objetos o valores que contendrá el arreglo, mínimo y máximo nos indican el límite inferior y el límite superior del rango de números entre los que se puede elegir. Se pueden repetir elecciones indefinidamente. (En caso de fraude) ← No leer

Ejemplo de un Array usando el método .rand

```
// se crea un Array de 5 valores aleatorios entre 0 y 10
Array.rand(5, 0, 10)   //  el resultado puede ser [0,5,8,6,8]
```

Notar que se eligió el número 8 dos veces

Los elementos de un arreglo tienen una posición que se indica por medio de un número llamado índice, El índice 0 corresponde al primer elemento del arreglo, el índice 1 corresponde al segundo elemento, el índice 2 al tercero y así sucesivamente.

Podemos pedir un elemento específico de un arreglo por medio del índice. Para hacer esto escribimos corchetes [ ] con el índice deseado después del arreglo.

```
x = [34, 51, 430, -2]     // definimos el arreglo x
x[0]  // pedimos el elemento en el índice 0 (e.g. el primer
elemento = 34)
x[1]  // pedimos el elemento en el índice 1 (e.g. el segundo
elemento = 51)
x[3]  // pedimos el elemento en el índice 0 (e.g. el cuarto
elemento = -2)
```

Funciones mas arreglos

Podemos juntar funciones con arreglos para obtener ideas estructuradas.
Ejemplo:

Creamos el arreglo *a* con 4 elementos:

```
a=[261.6, 329.6, 392, 523.2]
```

Podemos pedir sus elementos por índice:

```
a[0]  //  el primer elemento de a es 261.6
a[1]  //  el segundo elemento de a es 329.6
a[2]  //  el tercer elemento de a es 392
a[3]  //  el cuarto elemento de a es 523.2
```

Creamos una función que itere 4 veces la suma +1 arrojando los valores 0, 1, 2, 3

```
4.do{ | i | i.postln }
```

En la post aparece esto:

0
1
2
3
4

El 4 no fué un valor de i. El 4 se imprime por que la función se realizó 4 veces.

Juntamos la función con el arreglo *a* usando i como índice

```
4.do{ | i | a[i].postln }  // nos imprime uno por uno los
elementos de a en orden
```

261.6
329.6
392
523.2
4

El 4 no es parte del arreglo *a*. El 4 se imprime por que la función se realizó 4 veces.

Finalmente podemos hacer algo interesante: Usemos los elementos de *a* como argumentos de frecuencia para 4 SinOsc.

```
4.do{ | i | {SinOsc.ar(a[i], 0, 0.25)}.play }
```

La línea de código anterior produce lo mismo que las siguientes líneas, pero es más concisa y elegante.

```
(
{SinOsc.ar(261.6, 0, 0.25)}.play;
{SinOsc.ar(329.6, 0, 0.25)}.play;
{SinOsc.ar(392, 0, 0.25)}.play;
{SinOsc.ar(523.2, 0, 0.25)}.play;
)

// ================================
//  Pilón
// ================================

//  SuperCollider nos hace la tarea de crear la función
sinusoidal

{SinOsc.ar(100)}.plot

//  Pero nosotros tenemos ya las herramientas para hacerla

((2pi/100)*(0..100)).sin.plot
```