Tutoriel SuperCollider

Chapitre 2

Par Celeste Hutchins

2005

www.celesteh.com

Fonctions

Nicole a l'impression de maîtriser SuperCollider et a entendu Bush dire que l'économie repartait. Elle a donc abandonné ses études pour travailler dans la nouvelle start-up SuperCollider, SuperSounds.com. Le premier jour, son patron lui demande d'écrire une fonction qui imprime quatre fois "hello world". "Pas de problème", pense-t-elle, avant de consulter ses notes de cours. Les **fonctions** sont des blocs de code entourés de parenthèses, { } et hello world est assez facile à écrire. Elle écrit donc :

```
(
	{

		var greet ;
		greet = "hello world" ;

		saluer.postln
		;
		saluer.postln
		;
		saluer.postln
		;
		saluer.postln
		;
	}
)
```

Puis elle se dit : "J'aurais dû demander une prime à la signature". Elle essaie de l'exécuter en double-cliquant à droite de la parenthèse supérieure et en appuyant sur la touche "Entrée". Dans la fenêtre de sortie Untitled, il est écrit "`a Function`" (`une fonction`)

Quel est le problème ? Elle a déclaré une variable appelée `greet`. `greet` reçoit `"hello world"`. Elle envoie ensuite quatre fois un message `postln` à `greet`.

Chaque ligne se termine par un point-virgule . . .

Elle se rend alors compte qu'elle a défini une fonction, mais qu'elle n'a jamais demandé à l'interpréteur de l'exécuter. L'interpréteur a vu un bloc de code entouré de crochets et a pensé "une fonction !". Puis il s'est dit : "Qu'est-ce que tu veux que je fasse avec ça ? Rien ? Ok, je vais le jeter". Nicole modifie donc son code :

```
(
     var func ;

     func = {

          var greet ;
          greet = "hello world" ;

          saluer.postln
          ;
          saluer.postln
          ;
          saluer.postln
          ;
          saluer.postln
          ;
     } ;

     func.value ;
)
```

La `valeur` est un message que vous pouvez envoyer aux fonctions. Il signifie : "Exécutez-vous".

Mais Nicole reçoit un message de son patron disant : "Parfois, nous avons besoin d'imprimer cinq fois hello world, de temps en temps, trois fois, et rarement, il faut l'imprimer une infinité de fois". Nicole envisage d'écrire plusieurs versions différentes de la fonction et de les appeler func4, func3, etc, mais elle

se souvient alors des **arguments** des fonctions.

```
(
	var func ;
	func = { arg repeats ;

		var greet ;
		greet = "hello world" ;

		repeats.do ({
			saluer.postln ;
		}) ;
	} ;

	func.value(4) ;
)
```

Lorsqu'elle écrit sa fonction, elle **déclare** à l'interpréteur que la fonction prend un argument. Un argument est un type spécial de variable qui est défini lorsque la fonction est appelée. Lorsqu'elle appelle la fonction avec `func.value(4) ;`, elle attribue une valeur à l'argument `repeats`.

Ensuite, à l'intérieur de la fonction, elle a écrit : `repeats.do`. Qu'est-ce que "do" ? C'est un message. Il prend une fonction comme argument. 'do' est un message que vous pouvez envoyer à des **nombres** entiers et qui exécute la fonction passée en argument autant de fois que le nombre d'entiers sur lequel elle a été appelée. Dans cet exemple, repeats vaut 4, donc la fonction est exécutée quatre fois.

Qu'est-ce qu'un nombre entier ? Un nombre entier est un nombre entier. -2, -1, 0, 1, 2, 3, 4, etc. Il existe un nombre entier spécial dans SuperCollider appelé inf. Il signifie l'infini. Si nous essayons d'appeler notre fonction ci-dessus avec

```
func.value(inf) ;
```

hello world s'imprimera pour toujours, ou jusqu'à ce que nous arrêtions le programme en appuyant sur apple-. .

Puis le patron de Nicole envoie un autre courriel indiquant que le service marketing a procédé à quelques changements. Chaque ligne doit commencer par son numéro, en commençant par zéro. Nicole procède donc à une nouvelle modification :

```
(
     var func ;
     func = { arg repeats ;

          var greet ;
          greet = "hello world" ;

          repeats.do ({ arg index ;
               index.post ;
               " ".post ;
               greet.postln ;
          }) ;
     } ;

     func.value(4) ;
)
```

La fonction appelée par do prend un argument. Et cet argument est le nombre de fois que la boucle a été exécutée, en commençant par 0. La sortie de ce programme est donc :

```
0 Bonjour le monde
1 Bonjour le monde
2 Bonjour le monde
3 Bonjour le monde
```

`post` signifie simplement imprimer sans nouvelle ligne à la fin.

Presque à chaque fois qu'elle exécute cette fonction, l'argument sera 4. Elle peut donc déclarer un argument par défaut pour la fonction.

```
(
     var func ;
     func = { arg repeats = 4 ;

          var greet ;
          greet = "hello world" ;

          repeats.do ({ arg index ;
               index.post ;
               " ".post ;
               greet.postln ;
          }) ;
     } ;

     func.value ;
)
```

Lorsqu'elle appelle `func.value`, s'il n'y a pas d'argument, l'interpréteur attribue 4 à repeats par défaut. Si elle l'appelle avec `func.value(6)` ; alors repeats reçoit 6 à la place. Que se passe-t-il si elle passe accidentellement quelque chose d'autre qu'un entier dans sa fonction ? Cela dépend. Si l'objet qu'elle passe comprend également un message do, il l'utilisera à la place, bien que le résultat puisse être différent. Sinon, elle obtiendra une erreur.

Que se passe-t-il si la fonction prend beaucoup d'arguments avec des valeurs par défaut ?

```
(
	var func ;
	func = { arg foo = 0, bar = 0, baz = 1, repeats = 4 ;

		var greet ;
		greet = "hello world" ;

		repeats.do ({ arg index ;
			index.post ;
			" ".post ;
			greet.postln ;
		}) ;
	} ;

	func.value ;
)
```

Si elle veut passer des arguments, elle le fait dans l'ordre dans lequel ils sont déclarés.

```
func.value(0 /* foo */, 0 /* bar */, 1 /* baz */, 3 /* repeats */) ;
```

Cependant, si nous sommes satisfaits de toutes les valeurs par défaut, il existe un moyen de demander à la fonction d'affecter une variable particulière, dans le désordre :

```
func.value(repeats : 3) ;
```

Vous pouvez également vous contenter de passer les N premiers arguments et de laisser les valeurs par défaut pour le reste :

```
func.value(3, 1) ;
```

Il est également possible de combiner les approches :

```
func.value(2, repeats : 6) ;
```

Ces barres obliques sont des **commentaires**. L'interprète ignore tout ce qui se trouve entre une barre oblique et une barre oblique inverse. Ils peuvent s'étendre sur plusieurs lignes. Vous pouvez également créer un commentaire sur une seule ligne en utilisant une barre oblique avant :

```
// Il s'agit d'un commentaire
```

Il est parfois utile de **commenter** une ligne de code lors du **débogage**. Ainsi, vous pouvez sauter une ligne particulière afin de trouver l'origine de l'erreur.

L'objectif philosophique d'une fonction est de **renvoyer** une valeur. Faire des choses à l'intérieur d'une fonction, comme imprimer, est techniquement appelé **effet de bord**. La fonction renvoie la valeur de sa dernière ligne. Modifions cette fonction pour qu'elle renvoie le nombre de fois qu'elle a imprimé.

```
(
    var func ;
    func = { arg repeats = 4 ;

        var greet ;
        greet = "hello world" ;

        repeats.do ({ arg index ;
            index.post ;
            " ".post ;
            greet.postln ;
        }) ;
```

```
        se répète ;
    } ;

    func.value ;
)
```

Maintenant, si nous créons une nouvelle variable appelée `times`, nous pouvons lui affecter la sortie de la fonction.

```
(
    var func, times ;
    func = { arg repeats = 4 ;

        var greet ;
        greet = "hello world" ;

        repeats.do ({ arg index ;
            index.post ;
            " ".post ;
            greet.postln ;
        }) ;
        se répète ;
    } ;

    times = func.value ;
    times.postln ;
)
```

Imprime hello world avec le numéro de ligne comme précédemment, puis en bas, imprime un 4. Ou nous pourrions changer les deux dernières lignes de

```
    times = func.value ;
```

```
    times.postln ;
```

à

```
func.value.postln ;
```

et l'interprète lira cette déclaration de gauche à droite, en trouvant d'abord la valeur de la fonction et en envoyant ensuite cette valeur dans un message postln.

Ok, que se passe-t-il si nous prenons la fonction ci-dessus et que nous écrivons un code en dessous qui ressemble à ceci :

```
(
    var func, times ;
    func = { arg repeats = 4 ;

        var greet ;
        greet = "hello world" ;

        repeats.do ({ arg index ;
            index.post ;
            " ".post ;
            greet.postln ;
        }) ;
        se répète ;
    } ;
    saluer.postln ;
)
```

Nous obtenons des erreurs.

```
• ERREUR : Variable 'greet' non
   définie dans le fichier 'texte
   sélectionné'.
   ligne 14 char 6 :
     saluer-.postln ;
```

Cela est dû à ce que l'on appelle la **portée**. Les variables n'existent que dans le bloc de code dans lequel elles sont déclarées. Les blocs de code sont constitués de zéro ou plusieurs lignes de code entourées de parenthèses ou d'accolades. Cela signifie que les variables et les arguments déclarés à l'intérieur d'une fonction n'existent qu'à l'intérieur de cette fonction. `index` n'existe pas en dehors de sa fonction, qui est la fonction passée en argument à `repeats.do`. `greet` n'existe pas en dehors de sa fonction. Aucune de ces variables n'existe en dehors de la parenthèse.

Les variables des blocs extérieurs sont accessibles dans les blocs intérieurs. Par exemple,

```
(
     var func, times ;
     times = 0 ;
     func = { arg repeats = 4 ;

          var greet ;
          greet = "hello world" ;
          times.postln ;

          repeats.do ({ arg index ;
               index.post ;
               " ".post ;
               greet.postln ;
          }) ;
          se répète ;
     } ;
)
```

C'est bien parce que `times` existe dans le bloc le plus externe. De la même manière, nous pouvons utiliser greet dans notre fonction repeats.do.

Il existe cependant quelques variables qui peuvent être utilisées partout. L'interpréteur nous donne 26 **variables globales**. Leur portée est l'ensemble de SuperCollider. Les noms de ces variables ne comportent qu'une seule lettre : a, b, c, d, e, f, etc. Vous n'avez pas besoin de les déclarer et elles conservent leur valeur jusqu'à ce que vous la changiez à nouveau, même dans des blocs de code différents. C'est pourquoi la variable "s" fait référence au serveur. Vous pouvez la modifier, mais vous ne le souhaiterez peut-être pas.

Si vous avez des questions sur les fonctions, vous pouvez consulter le fichier d'aide des fonctions pour plus d'informations. Le message le plus utile que vous pouvez envoyer à une fonction est value, mais il en existe d'autres, que vous pouvez lire. Il se peut que vous ne compreniez pas tout ce que vous voyez dans les fichiers d'aide, mais il est bon de continuer à les lire.

Chiffres et mathématiques

Nous venons d'apprendre ce que sont les nombres entiers, qui, rappelons-le, sont des nombres entiers, comme -1, 0, 1, 2. Et nous avons appris ce qu'est le message do.

Quelles sont les opérations que l'on peut effectuer sur un nombre ? Ajouter, soustraire, multiplier, diviser, moduler.

Vous vous souvenez de l'algèbre où a*b + c*d = (a * b) + (c * d). Et vous vous souvenez que cette calculatrice bon marché ne faisait que ce que vous `tapiez`, dans l'ordre où vous le tapiez, sans respecter l'ordre des opérations ? SuperCollider est comme cette calculatrice bon marché. Les expressions mathématiques sont évaluées de gauche à droite. L'addition et la soustraction

ont la même **priorité** que la multiplication et la division. Cela signifie que

signifie que si vous voulez faire les choses dans un certain ordre, vous devez utiliser des parenthèses.

Les parenthèses sont évaluées de l'intérieur vers l'extérieur.

(a + (b * (c + d)))
(2 + (4 * (3 + 2))) = (2 + (4 *(5))) = (2 + (20)) = 22 ;

Une opération mathématique que vous n'avez peut-être jamais vue auparavant est le **module**. Il signifie "reste" et est représenté par un "%".

10 % 3 = 1
26 % 9 = 8

Ok, donc la sortie d'un module entre deux entiers est un entier, par définition. Et si vous ajoutez ou soustrayez deux entiers, vous obtenez un entier. Il en va de même si vous multipliez deux entiers. Mais quel est le résultat de la division ?

3 / 2 = 1.5

1,5 n'est pas un nombre entier. Il s'agit d'un type de nombre appelé virgule flottante ou, dans SuperCollider, un **flottant**. Un flottant est une fraction. 1,1, 2,5, -0,2, 5,0 sont tous des flottants. Ils peuvent également additionner, soustraire, etc.

L'histoire du numéro jusqu'à présent :

- Les nombres entiers sont des nombres entiers
- Les nombres réels sont des nombres flottants
- Indiquer l'ordre des opérations avec des parenthèses
- Les opérations mathématiques sont évaluées de l'intérieur vers l'extérieur et de gauche à droite.
- Le module (%) signifie le reste

Vous pouvez faire plus avec les nombres que de simples calculs mathématiques et des boucles d'exécution. Il existe de nombreux messages intéressants et utiles que l'on peut transmettre à Integer. Son fichier d'aide vaut la peine d'être lu. Tapez donc le mot Integer, commençant par un I majuscule, et appuyez sur apple- shift- ? pour consulter le fichier d'aide. Le début de ce fichier d'aide dit :

superclasse : SimpleNumber

superclasse est un mot de vocabulaire. Il fait référence à un concept appelé **héritage**. Cette désignation de superclasse signifie que Integer **est un** SimpleNumber. Lorsque vous définissez des classes (rappelez-vous qu'un appel est la définition d'un objet), vous pouvez définir des **sous-classes** de n'importe quelle classe. Une sous-classe est un type spécial de la classe d'origine. Elle **hérite de** toutes les propriétés de sa **superclasse**. La sous-classe est donc l'enfant et la superclasse est le parent. Nous y reviendrons. Mais ce que cela signifie pour nous, c'est que Integer **est un** SimpleNumber. Ce qui signifie qu'il comprend tous les messages que l'on peut passer à un SimpleNumber. Pour savoir quels sont ces messages hérités, nous devons mettre en surbrillance "SimpleNumber" et appuyer sur apple-shift- ? Float et Integer héritent tous deux de SimpleNumber, alors jetez un coup d'œil à ce fichier d'aide. Regarder les fichiers d'aide et essayer des choses vous permettra d'apprendre le langage plus rapidement que n'importe quoi d'autre. Il se peut qu'ils n'aient pas beaucoup de sens pour l'instant, mais si vous continuez à regarder, vous obtiendrez le contexte et serez capable de les comprendre à l'avenir.

Qu'en est-il de l'ordre des opérations à d'autres moments ? Si nous avons `func.value(3+4)`, il évalue `3+ 4` à `7` avant d'appeler la fonction. Nous pouvons mettre ce que nous voulons à l'intérieur de ces parenthèses et il évaluera ce qu'il y a à l'intérieur jusqu'à ce qu'il atteigne la parenthèse la plus éloignée, puis il appellera la fonction.

Nous avons encore un exemple qui, je l'espère, permettra de faire le lien entre tous ces éléments. Vous vous souvenez de notre SynthDef du chapitre 1 ?

```
(

      var syn, sound ;

      syn = SynthDef.new("example2", {arg freq = 440, amp = 0.2 ;
            Out.ar(0, SinOsc.ar(freq, mul : amp)) ;
      }) ;

      syn.load(s) ;
)
```

Nous avons ajouté un champ appelé "amp", qui indique l'**amplitude** ou le volume auquel le son doit être joué.

Maintenant, écrivons un code pour jouer N harmoniques de 100 Hz. Nous devons nous assurer que notre amplitude globale ne dépasse pas 1, sinon nous obtiendrons des crêtes. Nous allons donc diviser 1 par le nombre d'harmoniques en 1 pour obtenir la valeur d'amplification à envoyer à chaque synthé.

```
(
     var func ;

     func = { arg repeats = 4 ;
          repeats.do({ arg index ;

               Synth.new("exemple2", [\freq, (index + 1) * 100,
                                      \amp, 1 / répétitions]) ;
          }) ;
     } ;
     func.value ;
)
```

Écoutons ce qui se passe lorsque nous jouons ce morceau. Tous les sons sont joués en même temps. C'est parce que la boucle passe aussi vite qu'elle le peut. La prochaine fois, nous apprendrons à mettre en pause une boucle et à écrire un SynthDef qui s'arrête de jouer tout seul.

Problèmes

1. Traduisez les expressions algébriques suivantes dans la syntaxe appropriée du supercollisionneur, en utilisant des parenthèses si nécessaire. Vos réponses doivent être facilement compréhensibles, même pour les personnes qui ne connaissent pas l'ordre des opérations de SC.
   1.1. 3 + 5 * 4
   1.2. 3 * 2 + 1
   1.3. 3 * (4 + 2)
   1.4. 2 * 3 + 4 * 5 + 7 / 2 + 1

2. Familiarisez-vous avec le module en résolvant ces problèmes à la main :
   2.1. 5 % 4
   2.2. 163 % 9
   2.3. 20 % 5
   2.4. 17 % 6,
   2.5. 23 % 2
   2.6. 3 % 5

3. Ecrivez une fonction pour imprimer les *n* premiers multiples de 10, en commençant par 0. Passez le nombre de multiples dans la fonction en tant qu'argument. Fixez le nombre par défaut à 7.

4. `rand` est un message que vous pouvez envoyer à des nombres. `y.rand` renvoie un

nombre compris entre 0 et y. Les humains sont généralement capables d'entendre des fréquences comprises entre 20Hz et 20000Hz (l'argument freq est Hz). Écrivez une fonction pour jouer *n* hauteurs aléatoires et les *m* premières harmoniques de ces hauteurs. Assurez-vous que les hauteurs aléatoires se situent toutes dans la plage audible. Fixez l'amplitude des harmoniques à la moitié de l'amplitude des fondamentales. Transmettez le nombre de hauteurs aléatoires et le nombre d'harmoniques à la fonction en tant qu'arguments. Fixez le nombre de hauteurs aléatoires par défaut à 2 et le nombre d'harmoniques par défaut à 3. Assurez-vous que votre amplitude totale n'atteindra pas son maximum.

4.1. Pouvez-vous trouver un moyen de vous assurer que toutes vos hauteurs, y compris les harmoniques, se situent dans la plage audible ?