# MUS-1323 - Création musicale en langage Python

Olivier Bélanger

3 octobre 2019

# Table des matières

# Chapitre 1

# Guide d'installation

## 1.1 Installation des ressources essentielles

En premier lieu, pyo étant un module Python, une distribution de Python doit être installée sur le système. Ensuite, comme pyo comporte des éléments graphiques écrits avec WxPython, cette librairie doit aussi être installée sous la distribution de Python.

### 1.1.1 Installer Python

Installons d'abord la distribution officielle de Python, nous utiliserons la version 3.7.4, disponible à cette adresse :

https://www.python.org/downloads/release/python-374/

#### Windows

Sous Windows, installer la version qui correspond à l'architecture de votre ordinateur (fort probablement 64-bit), x86 pour un processeur 32-bit ou x86-64 pour un processeur 64-bit. Ce sont les fichiers :

32-bit : Windows x86 executable installer.

64-bit : Windows x86-64 executable installer.

**Attention** : Lors de l'installation, choisissez l'option « Customize Installation », puis laissez la configuration telle quelle mais assurez-vous de cocher la case « Add Python to environment variables ».

#### MacOS

Sous MacOS, installer la version pour 10.9 en montant, c'est le fichier identifié comme suit :

Mac OS X 64-bit installer.

Suivez les consignes d'installation !

### 1.1.2 Installer WxPython

Comme pyo comporte certains éléments d'interface graphique écrits avec **WxPython**, il faut maintenant installer cette librairie. L'installation se fait avec le gestionnaire de paquets intégré à Python : **pip**.

#### Windows

Ouvrez une invite de commande (*Command Prompt*, « cmd » dans la recherche) et entrez la ligne suivante, suivi de la touche « Retour » :

```
pip install -U wxPython
```

#### MacOS

Ouvrez l'application Terminal et entrez la ligne suivante, suivi de la touche « Retour » :

```
sudo pip3 install -U wxPython
```

### 1.1.3 Installer pyo

Pyo est aussi hébergé sur pypi.org et s'installe avec le gestionnaire de paquet officiel de python, **pip** :

#### Windows

Ouvrez une invite de commande et entrez la ligne suivante, suivi de la touche « Retour » :

```
pip install -U pyo
```

Un édtieur de texte spécialisé, **EPyo**, pour la création de script audio est installé en même temps que pyo. Si vous avez bien cocher la case « Add Python to environment variables » lors de l'installation de python, vous pouvez lancer l'éditeur en exécutant la commande suivante dans une invite de commande :

```
epyo
```

La plateforme Windows est particulièrement capricieuse en ce qui concerne les pilotes audio. Il y en a plusieurs et ils ne fonctionnent pas tous ! Si vous avez des problèmes avec la sortie audio de pyo, il s'agit de trouver le pilote qui fonctionne bien sur votre système. Référez-vous à la page de manuel sur l'inspection du système audio sous Windows pour vous guider :

http://ajaxsoundstudio.com/pyodoc/winaudioinspect.html

#### MacOS

Sous MacOS, pyo s'installe aussi via le gestionnaire de paquet officiel de python, **pip**. Dans une fenêtre de terminal, exécutez la commande suivante (vous aurez à fournir votre mot de passe) :

```
sudo pip3 install -U pyo
```

Un édtieur de texte spécialisé, **EPyo**, pour la création de script audio est installé en même temps que pyo. Vous pouvez lancer l'éditeur en exécutant la commande suivante dans une fenêtre de terminal :

```
epyo
```

#### linux

Sous les distributions majeures de linux, pyo s'installe aussi via le gestionnaire de paquet officiel de python, **pip**. Dans une fenêtre de terminal, exécutez la commande suivante :

```
sudo pip3 install -U pyo
```

### 1.1.4 Tester l'environnement de travail

Vous êtes maintenant prêt à exécuter un premier script pyo. Ouvrez l'application EPyo, sélectionnez un exemple dans le menu « Pyo Examples » (le fichier s'affichera dans l'éditeur de texte) et activez la commande « Run » en appuyant sur Ctrl+R (Cmd+R sous MacOS). La fenêtre du serveur audio devrait apparaître. Appuyer sur le bouton « Start » pour activer l'audio.

### 1.1.5 Documentation en ligne

Le manuel en ligne de pyo peut être consulté à l'adresse suivante :

http://ajaxsoundstudio.com/pyodoc/

# Chapitre 2

# Introduction au langage de programmation Python

## 2.1 Concepts généraux de la programmation

Un ordinateur ne fait rien d'autre que d'effectuer des opérations simples sur une séquence de signaux électriques. Ces signaux ne peuvent prendre que deux états, un potentiel électrique minimum ou maximum. On considérera généralement ces signaux comme une suite de nombres ayant comme valeur 0 ou 1. Un système numérique ainsi limité à deux chiffres est appelé système binaire.

Toute information d'un autre type devra donc être convertit au format binaire pour être traitée par l'ordinateur. Ceci est vrai non seulement pour tous les types de fichiers, tels que les textes, les images, les sons ou les films, mais aussi pour les différents programmes, c'est à dire les séquences d'instructions, qu'on voudra faire exécuter par l'ordinateur afin de traiter des données.

Ces nombres binaires forment une longue suite de 0 et de 1 et sont généralement traités par groupe de 8 (≪ octet ≫), 16, 32 ou 64, selon le système utilisé. Des traducteurs automatiques sont donc nécessaires afin de convertir les instructions constituées de chaînes de caractères, formant des mots-clés, en une suite de nombres binaires.

Le traducteur s'appellera **interpréteur** ou **compilateur**, selon la méthode utilisée pour effectuer la conversion. L'ensemble de mots-clés, associé à un ensemble de règles bien définies, constituera le **langage de programmation**. Les règles indiqueront au traducteur comment assembler les directives fournies par les mots-clés afin de composer des instructions compréhensibles par la machine, c'est à dire une suite de nombres binaires.

Selon le type d'instructions qu'il comporte, on dira d'un langage qu'il est de bas niveau (ex : Assembler) ou de haut niveau (ex : Pascal, Perl, Java, Python, ...). Un langage dit de bas niveau est constitué d'instructions très élémentaires, très proches du langage machine. Le langage de haut niveau offrira des commandes plus complètes et plus puissantes. Une seule commande dans un langage de haut niveau se traduira par plusieurs instructions élémentaires. Ce sera le rôle de l'interpréteur ou du compilateur de traduire ces instructions en langage machine. Les langages de haut niveau sont toujours beaucoup plus simples à apprendre et à utiliser puisqu'ils prennent en charge une grande partie du travail de programmation.

**Différents types de programmation :**

— **Programmation séquentielle** : Toutes les lignes du code source sont évaluées une à la suite des autres, de la première à la dernière. Pour exécuter de nouveau un bout de code ou sauter plusieurs lignes, il faut avoir recours à des instructions telles que **GOTO** ou **JUMP**. Ce type de programmation est très difficile à lire puisqu'il part dans toutes les directions. **Csound** ressemble en quelque sorte à de la programmation séquentielle.

— **Programmation procédurale** : Est basée sur la paradigme d'appel de procédure. Une procédure, aussi appelée *fonction*, *méthode* ou *routine*, consiste en une séquence de commandes à accomplir. Il est possible d'appeler n'importe quelle procédure à n'importe quelle étape du programme. Un des avantages de la programmation procédurale est la possibilité de réutiliser des bouts de code à plusieurs reprises sans avoir à les réécrire à chaque fois. Elle permet aussi la création de programmes modulaires et structurés, plus facile à lire que la programmation séquentielle. Les langages **Perl**, **Basic** et **C** sont des exemples de programmation procédurale.

— **Programmation orientée-objet** : La base de ce type de programmation est l'objet. Un objet est une structure de données répondant à un ensemble de messages. Les données qui décrivent la structure de l'objet sont appelés ses **attributs**, tandis que la réponse à la réception d'un message est appelée une **méthode**. Les objets sont définis par des classes, qui peuvent hériter des attributs et méthodes de d'autres classes. Ce type de programmation, plus moderne, est très puissante en ce qu'elle permet la compartimentation et la redéfinition de tous les éléments constituant le programme. **Java**, **C++** et **Python** sont des langages de programmation orienté-objet.

## 2.2 Introduction à Python

Python est un langage de programmation jeune, dynamique, gratuit et soutenu par une large communauté de programmeurs. Il est hautement extensible, c'est à dire que l'ajout de librairies, écrites en Python, C ou C++, est extrêmement simple. C'est un langage interprété, supportant l'approche modulaire et orientée objet de la programmation.

**Principales caractéristiques du langage :**

— Python est multi-plateforme, entièrement supporté sur les principaux systèmes d'exploitation (Mac OS, Windows, MS-DOS, Unix, etc).

— Python est gratuit et « open source », il est donc permis d'utiliser et de modifier le code source du logiciel.

— Python est efficace pour écrire de petits programmes de quelques lignes tout comme des programmes très complexes de plusieurs dizaines de milliers de lignes.

— Python offre une syntaxe extrêmement simplifiée, ainsi que des types de données hautement évolués (listes, dictionnaires et autres), permettant d'écrire des programmes très compacts,

lisibles et performants. Un programme Python est généralement de 3 à 5 fois plus court que le programme correspondant écrit en C.

— Python gère automatiquement ses ressources (mémoires, descripteurs de fichiers). Il n'est donc pas nécessaire de tenir des références et de s'assurer que les objets inutilisés sont bel et bien détruits.

— Les **pointeurs** n'existent pas en Python, ce qui simplifie grandement l'écriture des programmes.

— Python est un langage **orienté-objet** supportant l'héritage multiple et le polymorphisme, c'est à dire la réutilisation du même code avec des types de données différents.

— Le système de gestion des exceptions de Python est simple à utiliser et très efficace.

— Python est dynamiquement typé. Tout objet manipulable possède un type bien défini qu'il n'est pas nécessaire d'avoir préalablement déclaré.

— Python est un langage interprété. Les programmes sont compilés en instructions portables, puis exécutés par une machine virtuelle.

* Les deux dernières caractéristiques impliquent forcément un supplément d'opérations pour le processeur afin de faire fonctionner le programme correctement. Ceci se traduit par une vitesse d'exécution plus lente par rapport à un langage compilé.

## 2.3 Compilation et interprétation

Le programme écrit par le programmeur constitue le **code source** de l'application. Pour exécuter ce programme, le code source doit être traduit en code binaire exécutable par la machine. Comme il a été dit précédemment, il existe deux techniques pour effectuer cette traduction : l'interprétation et la compilation.

Lorsqu'un programme est interprété, aucun code objet n'est produit. L'interpréteur analyse chaque ligne du programme et la transforme en instructions du langage machine qui sont immédiatement exécutées.

$$\text{Code source} \implies \text{Interpréteur} \implies \text{Résultat}$$

La compilation d'un programme consiste à traduire à l'avance tout le code source et à créer un code objet, dont le langage est beaucoup plus proche de celui de la machine. Celui-ci devient un fichier exécutable en tout temps, sans le besoin de passer à nouveau par le compilateur.

$$\text{Code source} \implies \text{Compilateur} \implies \text{Code objet} \implies \text{Exécuteur} \implies \text{Résultat}$$

L'interprétation est idéale dans un contexte d'apprentissage ou de développement d'un programme, car elle permet de tester immédiatement les modifications apportées au programme sans avoir à compiler de nouveau le code source. Par contre, comme le langage doit être traduit en temps réel par l'interpréteur, certaines opérations peuvent demander beaucoup de temps et

ne pas être exécutées suffisamment rapidement au goût du programmeur. Dans ce cas, un programme compilé sera toujours l'option la plus rapide et efficace, puisque le code objet parle un langage beaucoup plus proche du langage machine.

Certains langages modernes, tel Python, tentent de faire le pont entre les deux techniques. Lorsque Python reçoit un programme source, il commence par le compiler pour produire un code intermédiaire (bytecode), plus près du langage machine, qui sera transmis à l'interpréteur pour l'exécution. L'interprétation du *bytecode* est plus rapide que celle du code source.

$$\text{Code source} \implies \text{Compilateur} \implies \text{Bytecode} \implies \text{Interpréteur} \implies \text{Résultat}$$

**Avantages :**

— Le *bytecode* est portable, il suffit d'avoir un interpréteur Python sur la machine pour exécuter n'importe quel code.

— L'interprétation du *bytecode* est suffisamment rapide, bien qu'elle soit plus lente que celle d'un véritable code objet, pour la grande majorité des applications.

— L'interpréteur est toujours disponible afin de tester rapidement des petits bouts de programme.

## 2.4 Les différents types d'erreurs

L'exécution d'un programme est une opération extrêmement capricieuse. La moindre petite erreur peut faire arrêter l'exécution. Trois types d'erreurs peuvent être rencontrées lors de la conception d'un programme :

— **Erreurs de syntaxe** : Tous les langages de programmation possèdent leur syntaxe propre. Celle-ci est généralement très précise et il est important de respecter les règles lors de l'écriture des programmes. La moindre faute de syntaxe, telle une virgule mal placée, provoquera l'arrêt du programme et l'affichage d'un message d'erreur. Il est très important de comprendre la structure des messages d'erreur, car ceux-ci nous donnent généralement des pistes très utiles pour corriger l'erreur qui a été commise. Par exemple, Python nous spécifiera si l'erreur est une erreur de syntaxe et à quelle ligne du programme elle a été détectée.

— **Erreurs sémantiques** : L'erreur sémantique, ou erreur de logique, se produit lorsque les instructions spécifiées sont valides mais ne correspondent pas à l'idée de départ du programmeur. Le programme s'exécute donc parfaitement, mais le résultat obtenu n'est pas celui qui était attendu. Ces erreurs sont plus compliquées à corriger. Il faut faire une analyse détaillée de toutes les parties du programme pour savoir quel bout de code ne donne pas le résultat escompté. Il existe plusieurs outils Python pour aider le *déboguage*, nous en détaillerons quelques-uns en temps et lieu.

— **Erreurs à l'exécution** : Ces erreurs se produisent principalement lorsqu'un programme, fonctionnant normalement, essaie d'exécuter une commande qui, pour une raison quelconque, n'est plus possible. Par exemple, le programme essaie de lire un fichier qui a été effacé du disque dur, ou il tente d'importer une librairie qui n'est pas installée sur le système. Ces erreurs se nomment des **exceptions**, et provoquent l'arrêt du programme. Nous verrons divers moyens de gérer les exceptions, afin de permettre à notre programme de continuer à rouler même lorsque cette situation arrive.

## 2.5 Noms des variables et mots réservés

Le travail d'un programme d'ordinateur consiste principalement à manipuler des données afin d'arriver à un certain résultat. Ces données sont mises en mémoire dans le langage à l'aide de variables. Ces variables peuvent être de différents types (que nous verrons ultérieurement) et portent un nom arbitraire, donné par le programmeur. Ces noms doivent être le plus évocateur possible, afin de faciliter la lecture du programme. Bien que l'on accède à une variable en l'appelant par son nom, il ne s'agit en fait que d'une référence désignant une adresse mémoire, c'est à dire un emplacement dans la mémoire vive de l'ordinateur. Cet emplacement contient la valeur réelle donnée à la variable sous un format binaire. N'importe quel objet peut être placé en mémoire, que ce soit un nombre entier, un nombre réel, une chaîne de caractères, une liste, un dictionnaire ou autre. Afin de distinguer ces différents contenus, le langage de programmation utilise des **types de variables**, « entier », « réel », « liste », etc. Ce sujet sera couvert dans le prochain chapitre.

Les noms donnés aux variables sont laissés au soin du programmeur. Ils doivent par contre respecter quelques règles, voici celles en vigueur dans le langage Python :

1. Un nom de variable peut être composé de lettres et de chiffres, mais doit toujours commencer par une lettre.
2. Le seul caractère spécial autorisé est le souligné (_). Les lettres accentuées, les cédilles, les espaces et les caractères spéciaux tels que #, $, @, etc, sont interdits. La raison est que ces caractères font déjà partie de la syntaxe de Python et ont un rôle particulier à jouer. Par exemple, le caractère # indique le début d'un commentaire, c'est à dire une ligne qui ne sera pas exécutée par le programme.
3. La casse (majuscule et minuscule) est significative. *Tempo* et *tempo* sont donc deux variables distinctes !

Par convention, les noms de variables commencent avec une minuscule et n'utilisent les majuscules que pour augmenter la lisibilité du code (ex : *listOfScales*). Il est très important qu'un nom de variable soit le plus évocateur possible, afin de rendre votre code plus lisible pour autrui.

Dans le langage Python (version 3), il y a 33 mots réservés qui ne peuvent être utilisés comme nom de variable puisqu'ils sont déjà utilisés par le langage lui-même et ont chacun une fonctionnalité propre. Ils sont listés dans le tableau de la page suivante.

| | | | | | |
|---|---|---|---|---|---|
| **and** | **as** | **assert** | **break** | **class** | **continue** |
| **def** | **del** | **elif** | **else** | **except** | **finally** |
| **for** | **from** | **global** | **if** | **import** | **in** |
| **is** | **lambda** | **nonlocal** | **not** | **or** | **pass** |
| **raise** | **return** | **try** | **while** | **with** | **yield** |
| **None** | **True** | **False** | | | |

## 2.6 Pratique

### 2.6.1 Premiers pas dans l'écriture d'un script audio

Tout script désirant utiliser les fonctionnalités de la librairie pyo doit d'abord importer le module. Nous reviendrons plus loin sur les différentes façons d'importer un module, pour l'instant nous allons utiliser la ligne suivante pour avoir un accès direct à toutes les classes et fonctions présentes dans la librairie. L'étoile signifie tous les éléments !

```
from pyo import *
```

Autre étape indispensable : Créer un serveur audio. Le serveur assure la communication audio avec la carte de son et c'est aussi lui qui gère l'horloge, c'est-à-dire le défilement des échantillons dans le temps. Un moteur audio fonctionne toujours par groupe d'échantillons, c'est-à-dire qu'il calcule tout un bloc d'échantillons pour un objet donné avant de calculer le bloc de l'objet suivant. La longueur de ce bloc est appelé *buffer size*. Le *buffer size* aura une incidence sur la charge en CPU ainsi que sur la latence du système, c'est-à-dire le délai induit par le calcul des blocs d'échantillons. Dans un premier temps, nous fonctionnerons avec les valeurs par défaut du serveur, que nous initialiserons comme suit :

```
s = Server().boot()
```

L'objet *Server* sans paramètre utilisera les valeurs définies par défaut. La méthode *boot* (nous détaillerons les classes et méthodes plus loin) indique au serveur d'initialiser la communication audio. L'horloge du serveur n'est pas encore démarrée mais il est maintenant prêt à accueillir des objets effectuant des processus sur le son. **Il est essentiel qu'un serveur existe avant de créer des objets pyo !**

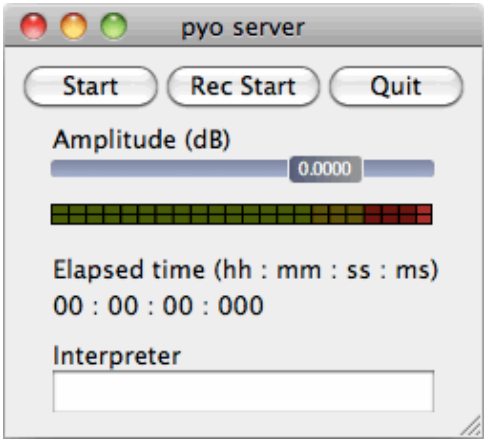
À partir du moment où un serveur est activé, tous les objets *pyo* qui seront créés seront automatiquement comptabilisés par le serveur, qui calculera les blocs d'échantillons selon l'ordre de création des objets. C'est ici que nous pouvons définir notre chaîne de traitement de signal. Commençons par l'exemple classique, une onde sinusoïdale à 1000 Hertz :

```
a = Sine().out()
```

Nous reviendrons dans quelques instants sur les notions de base des objets audio ! Nous avons maintenant un serveur prêt à calculer un signal, il ne reste qu'à démarrer son horloge interne. Nous utiliserons l'interface graphique incluse dans l'objet *Server* pour faciliter cette tâche. On appelle cette interface avec la méthode *gui* :

```
s.gui(locals())
```

À l'exécution du script, cette ligne provoquera l'affichage de la fenêtre offrant des contrôles de base sur le serveur, tel que le départ et l'arrêt de l'horloge, un bouton d'enregistrement sur le disque dur et un contrôle du volume global. Nous verrons en temps et lieu l'utilité d'ajouter *locals()* à l'intérieur des parenthèses.




Notre premier script ressemblera donc à ceci :

```
from pyo import *
s = Server().boot()
a = Sine().out()
s.gui(locals())
```

scripts/chapitre_03/01_play_sine.py

### 2.6.2 L'objet PyoObject

L'objet **PyoObject** est à la base (classe parente) de presque tous les objets disponibles dans la librairie. Les fonctionnalités de cet objet peuvent donc être appliquées à tous les objets qui en découlent (classe enfant). Plus tard, nous explorerons plus en détail l'héritage de classe. Pour l'instant, considérons que tout objet, si sa page de manuel contient la ligne suivante,

Parent : PyoObject

a accès aux fonctionnalités de cette classe.

#### Méthodes

Quelques méthodes qui s'applique à la majorité des objets de la libraire :

— **obj.play()** : La méthode *play* démarre le calcul des échantillons de l'objet donné. À noter que *play* n'envoie pas de son à la sortie audio.

— **obj.stop()** : La méthode *stop* arrête le calcul des échantillons de l'objet donné. Elle permet d'économiser du CPU si l'objet ne sert pas durant une certaine période.

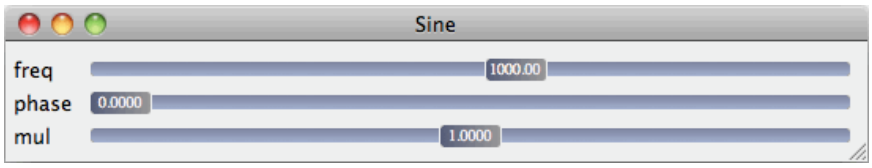
— **obj.out()** : La méthode *out* démarre le calcul des échantillons de l'objet donné et indique aussi que le signal généré par l'objet doit être acheminé à la sortie audio. Si un nombre entier est donné entre les parenthèses de la méthode, il indique le canal de sortie, en commençant à 0. Dans un système stéréo, 0 indique le canal de gauche et 1 indique le canal de droite.

— **obj.ctrl()** : La méthode *ctrl* affiche une fenêtre où des potentiomètres sont assignés aux paramètres de contrôle de l'objet donné. Cette fenêtre nous permettra, dans un premier temps, d'explorer rapidement l'impact des paramètres sur le signal sonore généré.

Dans le script précédent, insérer la ligne

```
a.ctrl()
```

juste après l'initialisation de l'objet *a*, affichera la fenêtre suivante :




### 2.6.3 Comment lire le manuel pyo

Les page du manuel sont séparés par objet. Chaque page peut contenir jusqu'à 6 sections. Voici une description sommaire de chaque section :

#### Section Titre

La première ligne suivant le nom de l'objet est toujours la ligne d'intitialisation, c'est-à-dire le moment où l'on crée un objet que l'on assigne à une variable. On y retrouve les paramètres régissant le comportement de l'objet ainsi que leur valeur par défaut, s'il y a lieu. Ensuite vient une description du processus effectué par l'objet, puis sa classe parente, s'il y a lieu. Si l'objet possède une classe parente, toutes les fonctionnalités de celle-ci s'appliquent à l'objet courant.

#### Section Paramètres

Description détaillée de chacun des paramètres de l'objet. Suivant le nom du paramètre, il y a toujours les types possibles pour ce dernier (ex : PyoObject, float, string, etc.). Il est très important de respecter les types permis pour un paramètre. S'il n'y a pas la mention *optional*, cela signifie qu'il est obligatoire de fournir une valeur au paramètre. Le paramètre *input* des objets qui transforment le signal (les effets) en est un bon exemple, l'objet attend un signal à transformer !

Paramètre obligatoire dont le seul type permis est le PyoObject :

```
input : PyoObject
    Input signal.
```

Paramètre optionnel acceptant les nombres réels (float) ou les PyoObject. Si le paramètre n'est pas spécifié, la valeur -7.0 sera utilisée :

```
transpo : float or PyoObject, optional
    Transposition factor in semitone. Defaults to -7.0.
```

### Section Notes

Si l'objet comporte certaines particularités qui lui sont propres, elles seront détaillées dans cette section. Par exemple, c'est ici que sera spécifié si un objet ne possède pas de méthode *out* (les signaux que l'on ne veut surtout pas envoyer dans les haut-parleurs) ou les attributs *mul* et *add*.

### Section Exemple

Cette section fournit un exemple d'utilisation de l'objet dans un processus simple. Le script peut être appelé en ligne de commande comme suit :

```
>>> from pyo import *
>>> example(Harmonizer)
```

N'hésitez pas à vous en inspirer !

### Section Méthodes

Ici sont listées toutes les méthodes permises sur un objet (auxquelles il faut bien sûr ajouter les méthodes de la classe parente !).

```
a = Harmonizer(input).out()
a.setTranspo(5)
```

*a.setTranspo(5)* remplace la valeur de transposition actuelle de l'objet et lui assigne 5.0.

### Section Attributs

Donne la liste des attributs d'un objet (en plus des attributs de la classe parente). Un attribut peut être appelé directement sur un objet sans passer par la méthode associée. Nous reviendrons plus tard sur l'utilisation des méthodes et des attributs.

L'exemple suivant effectue la même opération que l'exemple précédent.

```
a = Harmonizer(input).out()
a.transpo = 5
```

# Chapitre 3

# Opérations, types de variables et flux d'instructions

## 3.1 Calculer avec Python

Dans le langage Python, un nombre peut être exprimé sous différents formats. Il peut être soit entier, avec une résolution de 32 bits (ou plus si nécessaire) ou un nombre réel, souvent appelé nombre à virgule flottante.

— **Integer** : 12, -234, 0, ...
— **Float** : 0.005, 13.4654, -4.321, ...

Il faut garder en mémoire que la précision d'un nombre est dépendante de la quantité de bits utilisés pour représenter ce nombre. Dans le cas des nombres réels, la valeur utilisée sera presque toujours une approximation de la valeur demandée. Si vous tapez le nombre 3.23456 dans un interpréteur Python, voici ce que vous pourriez obtenir (le nombre de décimales à l'affichage dépend du système) :

```
>>> 3.23456
3.2345600000000001
```

Python est tout de même un environnement de calcul extrêmement précis, les nombres réels étant toujours représentés avec 64 bits. On constate que l'approximation est vraiment très proche de la valeur demandée. Python utilise les opérateurs standards pour exprimer les formules mathématiques :

— **Multiplication** : *
— **Division (nombres décimaux)** : / ( 5/2 = 2.5 )
— **Division (entier)** : / ( 5//2 = 2 ou int(5/2) = 2 )
— **Addition** : +
— **soustraction** : -

Avec Python 3, une division retourne toujours un nombre à virgule flottante. Le symbole // peut-être utilisé pour obtenir une division qui retourne un nombre entier. Ce symbole n'existe pas en Python 2, par contre, la syntaxe $int(5/2)$ est compatible entre les deux versions.

Deux autres opérateurs sont disponibles, l'exposant et le modulo.

— **Exposant** : ** ( 2**4 = 16 )
— **Modulo** : %

L'opérateur modulo retourne le restant d'une division d'un nombre par un autre.

```
>>> 3 % 2
1
>>> 12 % 5
2
>>> 12 % 6
0
```

Dans l'environnement Python, l'ordre de priorité standard est respecté. Il peut être mémoriser avec l'expression mnémonique « PEMDAS » (parenthèse, exposant, multiplication, division, addition, soustraction). Dans le cadre d'une expression complexe, les parenthèses sont toujours calculées en premier, ensuite ce sont les exposants, puis les multiplications et les divisions, qui ont le même ordre de priorité, et enfin les additions et les soustractions. Une simple erreur d'inattention dans l'ordre des priorités peut entraîner des résultats fort différents :

```
>>> 3.14159 * 4 ** 2.
50.26544
>>> (3.14159 * 4) ** 2.
157.913406496
```

## 3.2 Typage des variables

Dans plusieurs langages tels **C** ou **Java**, il est obligatoire de déclarer les variables que l'on compte utiliser, ainsi que leur type (c'est à dire que le typage est statique), avant de leur assigner un contenu. Une variable ne peut changer de type en cours d'exécution et n'acceptera pas une valeur qui ne correspond pas à son type. Voici un exemple en **C** où l'on a deux variables que l'on veut additionner ensembles :

```
int a = 5;
int b = 7;
int result;

result = a + b;
```

Sous Python, le typage des variables est dynamique, c'est à dire que Python se charge automatiquement d'attribuer le type qui correspond le mieux à toutes les variables spécifiées dans votre programme. La même opération en Python donnerait :

```
a = 5
b = 7
result = a + b
```

Le type « entier » sera automatiquement attribué aux variables *a* et *b* tandis que la variable *result* se verra attribuer un type selon le résultat de l'addition (dans cet exemple, un entier). Cette méthode est un peu plus lente à exécuter, puisque chaque fois que Python rencontre une variable, il doit déterminer son type. Par contre, elle est extrêmement flexible et permettra de gérer des structures de données hautement sophistiquées telles que les listes et les dictionnaires. Nous reviendrons plus tard sur ces types de données. Ainsi, sous Python, une variable peut très bien contenir un entier au début du programme et, en cours de route, devenir une liste ou un *string*.

## 3.3 Principaux types de données

En Python, on peut toujours questionner l'interpréteur à propos du type d'une variable donnée. Les principaux types de données disponibles dans le langage sont :

- **None** : Type de la valeur nulle *None*
- **int** : Un nombre entier
- **float** : Un nombre réel
- **str** : Une chaîne de caractères en format texte
- **bytes** : Une chaîne de caractères en format binaire
- **tuple** : Une liste entre parenthèse (1, 2, 3, 4)
- **list** : Une liste [1, 2, 3, 4]
- **dict** : Un dictionnaire

La fonction *type* permet de savoir quel est le type d'une variable :

```
>>> type(1)
<class 'int'>
>>> type(1.1)
<class 'float'>
>>> type("allo")
<class 'str'>
>>> type(b"allo")
<class 'bytes'>
>>> type((1,2))
<class 'tuple'>
>>> type([1, 2, 3, 4])
```

```
<class 'list'>
>>> type({"maj": [0, 4, 7]})
<class 'dict'>
>>> type(1.1) is float
True
>>> type("allo") is str
True
>>> type(b"allo") is bytes
True
```

Le type *string* sous Python possède une syntaxe particulière. Il peut être délimité par des guillemets simples ( ' ) ou des guillemets doubles ( " ). Les deux syntaxes sont tout à fait équivalentes mais cela permet d'introduire facilement un *string* dans un *string*!

```
>>> 'Ceci est un string'
'Ceci est un string'
>>> "Ceci est aussi un string"
'Ceci est aussi un string'
>>> 'Ceci est un "string" dans un string'
'Ceci est un "string" dans un string'
```

## 3.4 Affectation de valeurs

L'assignation d'une valeur à une variable s'effectue avec le symbole '='. Il ne s'agit en aucun cas d'une égalité dans le sens mathématique du terme. Cela veut simplement dire que nous avons déposé le contenu $x$ dans la variable $y$. Le nom de la variable représente un espace mémoire dans l'ordinateur où est enregistrée, en format binaire, la valeur affectée. Pour tester la relation d'égalité entre deux variables, on utilise le symbole '==', qui fait partie d'un groupe de symboles appelés **opérateurs de comparaison**, nous y reviendrons.

```
n = 10
```

Lorsque nous assignons une valeur à une variable, comme dans l'exemple ci-dessus, cette instruction a pour effet de réaliser plusieurs opérations dans la mémoire de l'ordinateur :

1. Créer et mémoriser un nom de variable
2. Lui attribuer un type bien déterminé
3. Créer et mémoriser une valeur particulière à l'endroit assigné
4. Établir un lien (par un système interne de pointeurs) entre le nom de la variable et l'espace mémoire de la valeur

**Affectations multiples**

Sous Python, on peut assigner la même valeur à plusieurs variables en même temps :

```
a = b = c = 1
```

Trois variables, $a$, $b$ et $c$, ont été créées et possèdent toutes la valeur 1.

**Affectations parallèles**

On peut aussi affecter plusieurs valeurs à plusieurs variables en même temps :

```
a, b = 0, 1
```

La valeur 0 a été affectée à la variable $a$ tandis que la valeur 1 a été affectée à la variable $b$.

## 3.5 Séquence d'instructions et exécution conditionnelle

### 3.5.1 Séquence d'instructions

Un programme informatique est toujours une série d'actions à effectuer dans un certain ordre. La structure de ces actions et l'ordre d'exécution constituent un algorithme. Les structures de contrôle sont des instructions qui déterminent l'ordre dans lequel les actions seront effectuées. Il existe trois types de structures : la **séquence**, la **sélection** et la **répétition**. Nous élaborerons sur la répétition au prochain cours.

Si votre programme ne contient pas d'instructions de **sélection** ou de **répétition**, les lignes seront exécutées dans l'ordre, de la première à la dernière. Lorsque Python rencontre une instruction conditionnelle, par exemple l'instruction **if**, il pourra prendre différents chemins selon le résultat de la condition.

### 3.5.2 Sélection ou exécution conditionnelle

L'exécution d'un programme complexe sera influencée en cours de route par différents facteurs, tels que les réponses fournies par l'utilisateur à certaines questions ou l'état de certaines variables du programme. C'est avec des instructions conditionnelles, placées à des endroits précis, que l'on spécifie au programme de prendre tel ou tel chemin selon l'état des données. Une instruction conditionnelle commence une ligne avec le mot clé **if** et la termine avec le symbole deux-points ( **:** ). Ce symbole marque le début d'une séquence de contrôle qui peut être une condition, une boucle, la définition d'une fonction ou d'une classe, etc. La ligne qui vient ensuite doit être indentée pour spécifier à Python que cette instruction ne doit être exécutée que dans certaines circonstances.

Par convention, l'indentation correspond à 4 espaces vides laissés au début de la ligne. Faites attention, la touche de tabulation, selon l'éditeur que vous utilisez, génère soit un caractère « TAB » (généralement équivalent à 4 espaces vides), soit 4 caractères d'espacement. Ces deux types de caractères ne sont pas encodés de la même façon par le langage et peuvent créer des conflits puisque Python ne considérera pas les lignes au même niveau d'indentation, même si visuellement tout parait impeccable. La plupart des éditeurs « intelligents » possèdent l'option de remplacer automatiquement les caractères « TAB » par 4 espaces vides.

Voici une instruction **if** simple, notez le changement de *prompt* (invite de commandes) de l'interpréteur Python au début de la ligne !

```
>>> a = 1
>>> if a == 1:
...     print('a vaut 1')
...
a vaut 1
```

Les trois points, que l'on appelle *prompt secondaire* signifient que nous entrons dans un bloc d'instructions que nous devons indenter. Comme la condition s'est avérée vrai, la ligne suivant celle du **if** a été exécutée et l'interpréteur a affiché *a vaut 1*. Si la condition s'était avérée fausse, cette ligne n'aurait tout simplement pas été exécutée.

Il est possible de spécifier un autre chemin, avec l'instruction **else**, pour le cas où une condition serait fausse :

```
>>> a = 0
>>> if a == 1:
...     print('a vaut 1')
... else:
...     print('a ne vaut pas 1')
...
a ne vaut pas 1
```

Notez bien les deux-points après les instructions **if** et **else** pour signifier l'entrée dans un bloc d'instructions à indenter. Notre programme définit maintenant deux chemins possibles en fonction de l'état de la variable *a* au moment d'exécuter ces lignes. L'instruction **elif** permet de complexifier une condition et de définir plusieurs actions différentes en fonction de plusieurs conditions :

```
>>> a = 0
>>> if a > 0:
...     print('a est positif')
... elif a < 0:
...     print('a est negatif')
... else:
...     print('a est ni positif ni negatif, a vaut donc 0')
...
a est ni positif ni negatif, a vaut donc 0
```

Une condition peut contenir autant de **elif** que nécessaire, mais une seule instruction **if** et une seule instruction **else**.

## 3.6 Opérateurs de comparaison

Les opérateurs de comparaison servent à tester la relation entre les différentes variables d'un programme. Selon le résultat obtenu, on peut décider de faire bifurquer le flux d'instructions pour opérer d'une certaine façon plutôt que d'une autre. Les différents opérateurs de comparaison sont :

| | |
|---|---|
| == | égalité |
| ! = | inégalité |
| < | plus petit que |
| > | plus grand que |
| <= | plus petit ou égal |
| >= | plus grand ou égal |

**Exemples** :

```
a = 10
if a <= 10:
    print('a est plus petit ou egal a 10')
elif a > 10:
    print('a est plus grand que 10')
else:
    print('Si un jour vous obtenez cette reponse, faites-le moi savoir!')
```

Plus tard dans la session, avec les connaissances que vous aurez acquises, reprenez ce script et essayez d'obtenir la réponse suivant l'instruction **else** !

```
a = 7
if (a % 2) == 0:
    print('a est un nombre pair')
    print('parce que le restant de sa division par 2 est 0')
else:
    print('a est un nombre impair')
```

Notez bien le signe ( == ) pour tester l'égalité entre deux valeurs. Le signe « égal » seul ( = ) est utilisé pour l'affectation d'une valeur à une variable.

## 3.7 Instructions composées

Le dernier exemple sur les opérateurs de comparaison constitue un premier bloc d'instructions, ou instructions composées. Sous Python, il y a deux règles essentielles à respecter pour construire un bloc d'instructions :

1. **La ligne qui annonce le bloc, l'en-tête, se termine toujours par un double point.** Les mots-clé suivants annonce un bloc d'instructions : *if*, *elif*, *else*, *for*, *while*, *def*, *class*, *with*, *try*, *except*, *finally*.
2. **Les lignes qui constituent le bloc d'instructions doivent être indentées exactement au même niveau (compter 4 espaces vides).**

```
a = 5
if a > 0: # ligne d'en-tete
    print('a est positif')
    print('car il est plus grand que 0')
else:
    print('a est soit negatif')
    print('soit nul')
    print('car il peut etre plus petit ou egal a 0')
```

Toutes les lignes d'un bloc d'instructions doivent absolument être au même niveau d'indentation. On appelle ces instructions un bloc logique, car chacune d'elle ne sera toujours exécutée que si la condition le permet. Quoiqu'il advienne, elles subiront toujours le même sort.

**Petit rappel** : n'oubliez pas que Python n'encode pas de la même façon un caractère de tabulation et un caractère d'espacement !

## 3.8 Instructions imbriquées

Afin de réaliser des structures décisionnelles complexes, vous aurez probablement à imbriquer les unes dans les autres plusieurs instructions composées.

```
if a >= 0:
    if (a % 2) == 0:
        if a < 100:
            print('a est une valeur pair entre 0 et 100')
        else:
            print('a est une valeur pair plus grande que 100')
            a = a % 100
    else:
        if a < 100:
            print('a est une valeur impair entre 0 et 100')
            a = a + 1
        else:
            print('a est une valeur impair plus grande que 100')
            a = (a % 100) + 1
else:
    print('a est negatif')
    a = 0
```

- Quelles sont les valeurs possibles pour la variable $a$ à la sortie de ce bloc d'instructions imbriquées ?

## 3.9 Règles de syntaxe

Sous Python, c'est la mise en page de votre programme qui définit les limites des blocs d'instructions. Cette règle, en plus de simplifier grandement la syntaxe, oblige le programmeur à écrire du code « propre », facile à lire pour quelqu'un qui n'en est pas l'auteur. Certains langages tels que le **C** ou le **Java** utilisent des symboles pour délimiter les blocs, généralement des accolades. Ceci permet de ne pas se soucier de l'indentation du programme et génère souvent du code pratiquement illisible, hormis pour le programmeur qui l'a écrit. En Python, ce sont les niveaux d'indentation qui délimitent les blocs constituants les conditions, les boucles, les fonctions et autres instructions composées.

```
# Bloc 1
....
ligne d'en-tete:
    # Bloc 2
    ....
    ligne d'en-tete:
        # Bloc 3
        ....
    # Bloc 2 (suite)
    ....
# Bloc 1 (suite)
....
```

En Python, une ligne d'instruction se termine avec le retour de chariot (ce qui évite l'emploi du point-virgule, rencontré dans de nombreux langages !). Nous verrons plus loin comment étendre une instruction sur plusieurs lignes. Il est aussi possible de terminer une ligne avec un commentaire. Un commentaire est tout ce qui se trouve entre le symbole dièse ( # ) et le retour de chariot. Ce texte sert de guide ou d'explication sur les différentes parties du programme et sera complètement ignoré par l'interpréteur lors de l'exécution.

Il est très important de bien commenter le programme à toutes les étapes de la construction, afin de vous y retrouver quelques années plus tard quand vous aurez à réorganiser votre code. Les commentaires sont des guides très précieux pour un programmeur qui, n'ayant pas participé au développement, voudrait apporter sa touche au programme. Il doit pouvoir comprendre rapidement quelles sont les directions que prend le flux d'instructions.

**À noter** : Un bon commentaire n'est pas celui qui dit comment c'est fait... mais plutôt celui qui dit pourquoi c'est fait ! Nous aurons l'occasion d'en rediscuter.

## 3.10 Exercice

### Calcul et condition

Sachant que la conversion de température s'effectue comme suit :

**C** à **F** $->$ on multiplie par 9, ensuite on divise par 5, puis on ajoute 32
**F** à **C** $->$ on soustraie 32, ensuite on multiplie par 5, puis on divise par 9

1. Assigner une valeur en Fahrenheit à une variable *f*.
2. Ensuite, effectuer une conversion de Fahrenheit à Celcius, assigner le résultat à une variable *c* et afficher la température en Celcius.
3. Puis, construire un bloc d'instructions conditionnelles qui affichera l'une des phrases suivantes, selon la température en Celcius :

   — Surement de la glace.

   — En etat d'ebulition.

   — Cette "eau" est ni trop chaude ni trop froide.

Attention, la précision mathématique est de rigueur ainsi que le respect de la syntaxe des phrases à afficher. De plus, votre script doit fonctionner impeccablement dans tous les cas de figure. Testez-le bien !

Tentez de réaliser l'exercice avant de consulter la solution en page 37.

## 3.11 Pratique

### 3.11.1 Créer une chaîne de traitement de signal

Tous les objets servant à modifier le signal (ex. : filtres et effets) ont un premier paramètre qui s'appelle *input*. Un objet de type *PyoObject* doit absolument être donné à ce paramètre. L'objet modificateur récupérera le signal de l'objet en *input* et lui appliquera un algorithme de transformation afin de généré son propre signal audio. Ce signal pourra être envoyé à la sortie audio, ou bien à un autre objet de transformation. Exemple :

```
from pyo import *
s = Server().boot()
fm = FM()
fm.ctrl()
filt = ButLP(fm).out(0)
filt.ctrl()
harmo = Harmonizer(filt).out(1)
harmo.ctrl()
s.gui(locals())
```

scripts/chapitre_03/02_dsp_chain.py

**Objets à explorer :**

— **Générateurs** : Sine, FM, SineLoop, LFO, Rossler, SuperSaw, SumOsc, RCOsc

— **Filtres** : ButLP, ButHP, ButBP, ButBR, Biquad, MoogLP, Reson, SVF

— **Effets** : Chorus, Disto, Freeverb, WGVerb, Harmonizer, FreqShift

— **Panoramisation** : Pan

Créer diverses chaînes de traitement, constituées d'un générateur suivi d'un ou plusieurs effets, et explorer avec les potentiomètres afin de vous familiariser avec le système et les sonorités. Attention aux volumes !

### 3.11.2 Instructions conditionnelles

Il est courant d'utiliser une instruction conditionnelle dans le but d'orienter un programme dans une direction particulière en modifiant seulement une variable. Par exemple, au lieu de modifier le programme à chaque nouvelle source que l'on veut explorer, on peut créer une instruction conditionnelle où chacune des sources correspondra à une condition particulière. Les conditions seront effectuées sur des nombres entiers :

```
src = 1

if src == 0:
    source = Sine().out()
elif src == 1:
    source = FM().out()
```

```
elif src == 2:
    source = SineLoop().out()

source.ctrl()
```

Selon la valeur que l'on donne à la variable *src* avant d'exécuter le script, la variable *source* représentera une onde sinusoïdale, une modulation de fréquence ou une onde sinusoïdale bouclée. Si on applique le même principe aux effets, on obtient :

```
from pyo import *
s = Server().boot()

src = 1
fx = 0

if src == 0:
    source = Sine().out()
elif src == 1:
    source = FM().out()
elif src == 2:
    source = SineLoop().out()

source.ctrl()

if fx == 0:
    effet = Chorus(source).out(1)
elif fx == 1:
    effet = Harmonizer(source).out(1)
elif fx == 2:
    effet = Disto(source).out(1)

effet.ctrl()

s.gui(locals())
```

scripts/chapitre_03/03_conditions.py

### 3.11.3 Pages de manuel à consulter

| | | | |
|---|---|---|---|
| **Server** | **PyoObject** | **Sine** | **FM** |
| **SineLoop** | **LFO** | **SuperSaw** | **SumOsc** |
| **RCOsc** | **ButLP** | **ButHP** | **ButBP** |
| **ButBR** | **Biquad** | **MoogLP** | **Reson** |
| **SVF** | **WGVerb** | **Chorus** | **Disto** |
| **Freeverb** | **Harmonizer** | **FreqShift** | **Pan** |

### 3.11.4 Exemples et exercices

Tous les exercices proposés dans ces pages ont pour but de vous guider dans l'élaboration d'une librairie personnelle de processus sonores que vous utiliserez lors de la réalisation des travaux. Il est donc très important de suivre les étapes consciencieusement, afin de bien maîtriser

les différentes techniques, et de conserver tous vos essais. Vous devrez régulièrement reprendre ces bouts de code pour les étoffer.

La seule et unique recette efficace pour l'apprentissage d'un langage de programmation est la **pratique**. Prendre le temps nécessaire pour comprendre et assimiler ces exercices simples fera en sorte que la réalisation de processus plus complexes (et de ce fait, plus intéressant !) vous semblera beaucoup plus aisée.

**Exploration de sonorités à partir de sources élémentaires de synthèse.**

**Exercices** :

En utilisant les objets proposés précédemment (générateurs, filtres et effets de la page « Premier pas »), explorer diverses chaînes de traitement de votre composition.

1. Vous pouvez créer autant d'objets que nécessaires, c'est-à-dire plusieurs sources et/ou plusieurs effets connectés en cascade ou en parallèle.
2. Utilisez les valeurs par défaut pour l'initialisation des objets. Le seul argument à spécifier est l'*input* audio pour les effets. Vous pouvez aussi spécifier un canal de sortie à la méthode *out()*. Exemple, *out(1)* pour acheminer le signal vers le haut-parleur de droite.
3. Affichez la fenêtre de contrôle (avec la méthode *ctrl()*) pour varier la valeur des différents paramètres.
4. Attention au volume ! Le comportement par défaut des objets est de jouer à pleine puissance. Pour l'instant, nous allons simplement réduire le volume du serveur, qui agit sur le signal final. Observez comment l'exemple suivant initialise le serveur à une amplitude de -20 dB !
5. Vous êtes libre de garder toutes vos expérimentations dans un seul script et de naviguer d'une à l'autre à l'aide d'une condition (comme dans l'exemple suivant) ou de travailler sur des fichiers indépendants, c'est selon votre aisance.

**Exemples** :

Le script suivant propose trois chaînes de traitement avec des schémas de connexion différents. Le premier exemple est constitué d'une source et de deux effets connectés en cascade où l'on écoute que le résultat final. Le deuxième propose une source connectée en parallèle à deux effets (chacun des effets est acheminé vers un haut-parleur) et le dernier est bâti avec trois sources additionnées en entrée d'un seul effet.

```
"""
Exemples de chaines de traitement. Aucun parametres a l'initialisation.
Exploration avec la fenetre 'ctrl'.

Exemple no 1 : Connections en cascade :
    sine  ->  disto  ->  chorus

Exemple no 2 : Connections en parallele (multi-effets) :
    sineloop  ->  freqshift
              ->  freqshift

Exemple no 3 : Connections en parallele (multi-sources) :
    lfo 1  ->
    lfo 2  ->  harmonizer
    lfo 3  ->

"""
from pyo import *

s = Server().boot()
s.amp = .1 # initialise le serveur a -20 dB

exemple = 2 # modifier cette variable pour changer d'exemple

if exemple == 1:
    src = Sine()
    src.ctrl()
    dist = Disto(src)
    dist.ctrl()
    chor = Chorus(dist).out()
    chor.ctrl()
elif exemple == 2:
    src = SineLoop()
    src.ctrl()
    fshif = FreqShift(src).out()
    fshif.ctrl()
    fshif2 = FreqShift(src).out(1)
    fshif2.ctrl()
elif exemple == 3:
    src1 = LFO().out()
    src1.ctrl()
    src2 = LFO().out()
    src2.ctrl()
    src3 = LFO().out()
    src3.ctrl()
    harm = Harmonizer(src1+src2+src3).out(1)
    harm.ctrl()

s.gui(locals())
```

scripts/chapitre_03/04_examples.py

### 3.11.5 Solution à l'exercice de conversion de température

```
f = 212

c = (f - 32) * 5. / 9

print(c)

if c <= 0:
    print('Surement de la glace')
elif c >= 100:
    print("En etat d'ebulition")
else:
    print('Cette "eau" est ni trop chaude ni trop froide')
```

scripts/chapitre_03/05_exercice.py

# Chapitre 4

# Les structures de données

## 4.1 Chaînes de caractères : fonctions propres au type *string*

Un *string* est une constante, c'est à dire qu'il est immuable (il ne peut être modifié). Pour modifier un *string* il faut obligatoirement en créer un nouveau. Il peut être délimité soit par des guillemets simples (’), soit par des guillemets doubles (”).

```
s1 = 'Ceci est un string'
s2 = "Ceci est aussi un string"
```

Les guillemets simples et doubles sont équivalents, mais peuvent servir à introduire un *string* dans un *string*, comme dans l'exemple ci-dessous :

```
>>> print('Ceci est un "string" dans un string')
Ceci est un "string" dans un string
```

### 4.1.1 Opérations

Comme pour l'affectation d'un nombre, l'affectation d'un *string* à une variable s'effectue avec le symbole « égal » ( = ) :

```
str = 'Ceci est un string'
```

La concaténation de deux *strings*, pour créer un seul *string* contenant les deux éléments, s'effectue avec le symbole « plus » ( + ) :

```
>>> str = 'Hello ' + 'world!'
>>> str
'Hello world!'
```

On peut créer un *string* qui contient plusieurs répétitions d'un autre *string* avec le symbole « étoile » ( * ) :

```
>>> str = 'allo' * 3
>>> str
'alloalloallo'
```

### 4.1.2 Construction d'un string avec le symbole pourcentage (%)

Si le symbole % apparaît dans un *string* suivi d'une des lettres *d*, *f*, *s*, *e* ou *x*, cela indique que l'on veut remplacer ce symbole par une variable de notre script. Il faudra donc faire suivre le *string* d'un autre symbole %, puis de la variable ou des variables que l'on désire introduire dans le *string*. Si plus d'une variable doivent être spécifiées, il faut les placer entre parenthèses.

Les lettres après le symbole % signifient que l'on veut introduire :

d = un entier
f = un nombre réel
s = un *string*
e = un nombre en format exponentiel
x = un nombre en format hexadécimal

```
>>> a, b, c = 1, 3, 7
>>> print('%d, %d et %d sont des nombres premiers' % (a, b, c))
1, 3 et 7 sont des nombres premiers
>>>
>>> a, b, c = 1.02, 3.2, 7.67
>>> print('%f, %f et %f sont des nombres reels' % (a, b, c))
1.020000, 3.200000 et 7.670000 sont des nombres reels
>>>
>>> a, b, c = 'Bill', 'Bob', 'Joe'
>>> print('%s, %s et %s sont probablement americains' % (a, b, c))
Bill, Bob et Joe sont probablement americains
```

À noter que les nombres réels ont 6 valeurs décimales par défaut. On peut modifier ce nombre en insérant un point, puis le nombre de décimales que l'on désire, entre le signe % et la lettre f :

```
>>> a, b, c = 1.0212, 3.2, 7.67
>>> print('%.2f, %.2f et %.2f sont des nombres reels' % (a, b, c))
1.02, 3.20 et 7.67 sont des nombres reels
```

Ceci peut aussi s'appliquer pour indiquer avec combien de chiffres on veut représenter un entier ou pour restreindre le nombre de caractères possibles à l'insertion d'un chiffre dans un *string*.

### 4.1.3 Quelques méthodes de la classe string

**upper**() : convertit tous les caractères en majuscules

```
>>> a = 'allo'
>>> a = a.upper()
>>> a
'ALLO'
```

**lower**() : convertit tous les caractères en minuscules

```
>>> a = 'ALLO'
>>> a = a.lower()
>>> a
'allo'
```

**capitalize**() : Assure que la première lettre est une majuscule

```
>>> a = 'allo'
>>> a = a.capitalize()
>>> a
'Allo'
```

**replace**(*old*, *new*) : remplace toutes les occurrences de *old* par *new*.

```
>>> a = 'allo'
>>> a = a.replace('l', 'b')
>>> a
'abbo'
```

**find**(*sub*) : retourne l'index de la première occurrence de *sub*, retourne -1 si *sub* n'existe pas.

```
>>> a = 'Bonjour tout le monde!'
>>> a.find('jour')
3
```

**join**(*seq*) : retourne un *string* qui contient la concaténation de tous les éléments de *seq*. L'élément séparateur est le *string* sur lequel est appliquée la méthode.

```
>>> a = 'allo'
>>> b = ' '.join(a)
>>> b
'a l l o'
```

**split**(*sep*) : retourne une liste des mots contenus dans le *string* en utilisant *sep* comme séparateur.

```
>>> a = 'Bonjour tout le monde!'
>>> b = a.split(' ')
>>> b
['Bonjour', 'tout', 'le', 'monde!']
```

Pour le détail des méthodes de la classe *string*, se référer à la documentation de Python.

### 4.1.4 Indiçage

On peut facilement obtenir un caractère ou une chaîne de caractères contenu dans un *string* avec la méthode d'indiçage. Comme pour tous les éléments considérés comme des séquences (*string*, liste, tuple), on accède aux éléments de la séquence en spécifiant entre crochets l'index que l'on désire obtenir. Une valeur simple retournera le caractère à cette position dans le *string*, tandis que 2 valeurs séparées par les deux-points (par exemple [2:8] ) retournera un *string* contenant la sous-chaîne de caractères spécifiée :

```
>>> a = 'Bonjour tout le monde!'
>>> a[0]
'B'
>>> a[5]
'u'
>>> a[2:10]
'njour to'
>>> a[:10] # du debut jusqu'a la position 10
'Bonjour to'
>>> a[16:] # de la position 16 jusqu'a la fin
'monde!'
>>> a[-1] # derniere position
'!'
>>> a[:-4] # du debut a la fin moins les 4 dernieres positions
'Bonjour tout le mo'
```

### 4.1.5 La méthode *len*

**len**(*string*) : la fonction len() retourne le nombre de caractères, incluant les espaces vides, dans un *string* :

```
>>> a = 'Bonjour tout le monde!'
>>> len(a)
22
```

### 4.1.6 *String* de documentation

Un *string* entre guillemets triples (' ' ' ou """) à la ligne suivant la déclaration d'une fonction sera enregistré dans la variable __doc__ de la fonction.

```
>>> def puissance(x, exp=2):
...     '''Calcule et retourne la valeur de x a la puissance exp'''
...     return x**exp
...
>>> puissance.__doc__
'Calcule et retourne la valeur de x a la puissance exp'
```

## 4.2 Les listes : Opérations sur les listes

Une liste consiste simplement en une suite de valeurs séparées par des virgules et placées entre crochets [ ]. Les éléments dans une liste peuvent être de différents types (nombre, string, liste, tuple, dictionnaire, fonction, objet, ...) et peuvent être mélangés. La liste est altérable, c'est à dire que l'on peut modifier ses éléments sans avoir à créer une nouvelle liste.

```
>>> a = [3, 5, 2, 8, 7, 5, 1]
>>> a.sort()
>>> a
[1, 2, 3, 5, 5, 7, 8]
```

### 4.2.1 Quelques opérations sur les listes

- **list1 + list2** : concaténation de list1 et list2
- **list[i]** : retourne l'élément à la position i
- **list[i : j]** : retourne une liste contenant les éléments entre les positions i et j
- **len(list)** : retourne la longueur de la liste
- **del list[i]** : élimine l'élément à la position i
- **list.append(val)** : ajoute l'élément *val* à la fin de la liste
- **list.extend(list)** : ajoute une liste à la fin d'une liste
- **list.sort()** : met dans l'ordre les éléments d'une liste
- **list.reverse()** : inverse les éléments d'une liste
- **list.insert(i, val)** : insère *val* à la position i
- **list.count(val)** : retourne le nombre d'occurrences de *val* dans une liste
- **list.pop()** : retourne et élimine la dernière valeur d'une liste
- **val in list** : retourne **True** si l'élément *val* est présent dans la liste

Très bon tutoriel sur les listes dans la documentation Python :

## 4.3 La fonction *range*

La fonction **range** génère un itérateur de nombres entiers qui peut être très utile pour créer rapidement une liste d'entiers (Sous python 2, la fonction **range** retourne directement une liste). Elle permet, entre autre, de boucler sur un certain nombre de valeurs à l'intérieur d'une boucle **for**.

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> for i in range(5):
...     print(i, end=" ")
...
0 1 2 3 4
```

Nous élaborerons sur la construction des instructions répétitives au prochain cours.

#### Arguments de la fonction range

Si un seul argument est donné à la fonction *range*, il sera considéré comme la valeur maximum et la liste commencera à 0. Si deux arguments sont donnés à la fonction, ils représentent la valeur minimum et la valeur maximum. Ce qui générera une liste de la valeur minimum (incluse) à la plus grande valeur possible plus petite que la valeur maximum. Un troisième argument peut être donné pour spécifier le 'pas', c'est à dire de combien de valeurs on avance à chaque élément de la liste.

```
>>> list(range(2, 10))
[2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2, 10, 2))
[2, 4, 6, 8]
>>> list(range(10, 50, 5))
[10, 15, 20, 25, 30, 35, 40, 45]
```

## 4.4 Pratique

### 4.4.1 Contrôle des objets via les paramètres d'initialisation

Nous venons tout juste de voir comment modifier le comportement de la fonction *range* en lui donnant des arguments entre les parenthèses. Le comportement des objets de la librairie *pyo* peut également être régit de cette façon. Il y a cependant quelques règles à respecter :

**1 - Si les noms des arguments ne sont pas spécifiés, ces derniers doivent être donnés dans l'ordre.**

Suivant cette méthode, considérant que l'ordre des arguments de l'objet *Sine* est :

freq, phase, mul, add

Pour assigner une valeur d'amplitude de 0.5 (argument *mul*), on doit spécifier tous les arguments qui le précède :

```
a = Sine(1000, 0, .5)
```

**2 - Certains arguments peuvent être omis ou l'ordre peut être mélangé seulement si on spécifie le nom des arguments en leur affectant une valeur.**

```
a = Sine(mul=.5)
b = Sine(mul=.3, freq=500)
```

**3 - On peut mélanger les 2 méthodes, mais dès qu'un argument est dûment nommé, l'ordre est considéré comme étant brisé et l'on doit spécifier le nom pour le reste des arguments.**

```
a = Sine(mul=.5)
b = Harmonizer(a, -12, mul=.3, feedback=0.5)
```

Le manuel devient donc un outil indispensable lors de l'écriture d'un scripts. Vous pouvez consulter le manuel en ligne si vous êtes connecté sur le web ou bien utiliser le système d'aide intégré de Python en appelant la méthode *help* :

```
>>> from pyo import *
>>> help(Sine)

Help on class Sine in module pyolib.generators:
...
```

La librairie *pyo* fournit aussi une fonction *class_args* permettant d'obtenir rapidement les paramètres d'initialisation d'un objet ainsi que leur valeur par défaut :

```
>>> from pyo import *
>>> class_args(Sine)
'Sine(freq=1000, phase=0, mul=1, add=0)'
```

Une bonne habitude est de toujours avoir un interpréteur ouvert lorsque l'on écrit un script ! En prime, vous pouvez écouter les exemples du manuel avec la fonction *example* :

```
>>> from pyo import *
>>> example(Sine)

import time
from pyo import *
s = Server().boot()
s.start()
sine = Sine(freq=500).out()
time.sleep(5.000000)
s.stop()
time.sleep(0.25)
s.shutdown()
```

### 4.4.2 Lire un fichier son sur le disque dur

Nous allons maintenant expérimenter avec l'objet *SfPlayer* qui permet de lire un fichier son sauvegardé sur le disque dur.

```
>>> from pyo import *
>>> class_args(SfPlayer)
'SfPlayer(path, speed=1, loop=False, offset=0, interp=2, mul=1, add=0)'
```

L'argument *path* est obligatoire puisque qu'aucune valeur ne lui est assignée par défaut. On spécifie le « chemin » vers le son à l'aide d'un *string* en utilisant la syntaxe en vigueur sur les systèmes Unix, c'est-à-dire en séparant les éléments hiérarchiques par le symbole « / ».

"/Users/olivier/Desktop/snds/baseballmajeur_s.aif"

```
>>> from pyo import *
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(path="/Users/olivier/Desktop/snds/baseballmajeur_s.aif").out()
```

Sur la plateforme Windows, python se chargera de convertir votre *string* dans le bon format avant d'aller chercher le son sur le disque. Nous y écrierions donc :

"C :/Documents and Settings/olivier/Desktop/snds/baseballmajeur_s.aif"

Les paramètres optionnels permettent de contrôler respectivement :

- **speed** : La vitesse de lecture (0.5 = deux fois plus lent, 2 = deux fois plus vite)
- **loop** : Si le son est lu en boucle ou non (*True* = lu en boucle, *False* = lu une seule fois)
- **offset** : Le point de départ, en secondes, dans le fichier son (2 = la lecture commence à 2 secondes dans le fichier)
- **interp** : La qualité de l'interpolation à la lecture (n'y touchez pas pour l'instant !)
- **mul** : Le facteur d'amplitude (0.5 = deux fois moins fort)
- **add** : La valeur à ajouter à chacun des échantillons après la lecture (À ne pas modifier dans le cas d'une lecture de fichier son sous risque d'endommager vos haut-parleurs et vos oreilles !)

**Exercice**

Faire jouer un son en boucle et lui appliquer une chaîne de traitements sonores choisis parmi ceux introduits au cours précédent. Gérer les paramètres des effets via les arguments à l'initialisation.

### 4.4.3 Gestion de la polyphonie

Si, à l'exercice précédent, vous avez chargé un son stéréo, vous avez pu remarquer que du signal était envoyé aux deux haut-parleurs. Ceci est du au fait que chaque objet de la librairie *pyo* est en mesure de gérer plusieurs *streams* audio à la fois. Un *stream* est un objet interne de la librairie chargé de véhiculer un canal de son monophonique. Chaque objet gère son audio via

un ou plusieurs *streams*, qui sont récupérés par le serveur et acheminés vers une sortie de la carte de son. La fonction *len*, applicable aux listes, peut aussi être appelée avec un objet *pyo* en argument. Elle retournera le nombre de *streams* gérés par l'objet.

```
>>> from pyo import *
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(path="/Users/olivier/Desktop/snds/baseballmajeur_s.aif").out()
>>> len(a)
2
```

En ce qui concerne la distribution des *streams* d'un objet vers les sorties audio, le comportement par défaut (modifiable via les arguments de la méthode *out*) est d'alterner la sortie, en fonction du nombre de sorties disponibles, à chaque nouveau *stream*. En débutant à 0, un son stéréo sera répartit sur les sortie 0 et 1, c'est-à-dire le premier *stream* à gauche et le second à droite.

**Arguments de la méthode out**

- **chnl** : Le canal où sera envoyé le premier *stream* de l'objet.
- **inc** : De combien de canaux on avance à chaque nouveau *stream*.
- **dur** : Durée d'activation, en seconde, de l'objet. La méthode *stop* est appelée automatiquement lorsque le temps est écoulé. Une durée de 0 signifie une durée infinie.
- **delay** : Délai, en seconde, avant l'activation de l'objet.

Dans un environnement multi-canaux, ces paramètres peuvent être manipulés de façon très subtile. En stéréo, les valeurs par défaut (chnl=0, inc=1) sont généralement les plus cohérentes ! On peut convertir le nombre de canaux d'un objet avec la méthode *mix* comme suit :

```
>>> a = Sine()
>>> len(a)
1
>>> b = a.mix(2)
>>> len(b)
2
>>> print(b)
< Instance of Mix class >
>>> c = SfPlayer("/Users/olivier/Desktop/snds/baseballmajeur_s.aif")
>>> len(c)
2
>>> d = c.mix(1)
>>> len(d)
1
>>> print(d)
< Instance of Mix class >
```

Notez que *b* et *d* sont de nouveaux objets de la classe *Mix* qui ont été créés par l'appel de la méthode *mix*. Ils reçoivent les signaux de *a* et *c*, effectuent la conversion et gèrent de nouveaux *streams*... *a* et *c* restent intacts.

### 4.4.4 Utilisation des listes à des fins musicales

Chaque objet de la librairie *pyo* prend un certain nombre d'arguments afin de définir son comportement. Par exemple, le paramètre « path » de l'objet *SfPlayer* prendra un lien vers un fichier son sur le disque dur. Rappelons la ligne pour jouer un son à partir du disque dur :

```
a = SfPlayer("/home/olivier/Dropbox/private/snds/baseballmajeur_s.aif")
```

Un concept très puissant en programmation, appelé *multi-channel expansion*, consiste en la possibilité d'utiliser une liste comme valeur à un argument. C'est un concept généralisé à tous les paramètres des objets *pyo* (sauf mention contraire dans le manuel!), afin de décupler les processus sonores sans avoir à répéter sans cesse les mêmes lignes de code. Ainsi, pour créer un décalage progressif de la lecture d'un son, on exécuterait le code suivant.

```
from pyo import *
s = Server().boot()
# Dans un path, le symbole ".." recule d'un dossier dans la hierarchie.
# Un path relatif commence au dossier ou est sauvegarde le script python.
a = SfPlayer(path="../snds/baseballmajeur_s.aif",
             speed=[1,1.005,1.007,.992], loop=True, mul=.25).out()
s.gui(locals())
```

scripts/chapitre_04/01_snd_chorus.py

La fonction *range*, vue un peu plus tôt, pourrait servir à créer une suite d'harmoniques :

```
from pyo import *
s = Server().boot()
a = Sine(freq=list(range(100,1000,100)),
         mul=[.5,.35,.2,.14,.1,.07,.05,.03,.01]).out()
s.gui(locals())
```

scripts/chapitre_04/02_serie_harmonique.py

Un objet qui reçoit une liste comme valeur d'un de ses arguments créera le nombre de *streams* nécessaires afin de calculer le rendu sonore désiré. Si, pour un même objet, plusieurs arguments reçoivent une liste en entrée, la plus longue liste sera utilisée pour générer le nombre de *streams* nécessaires. Les valeurs des listes plus courtes seront utilisées dans l'ordre, avec bouclage lorsque la fin de la liste est atteinte. Utiliser des listes de différentes longueurs pour les arguments d'un objet produira un résultat sonore riche et varié puisque les combinaisons de paramètres seront différentes pour chaque instance du processus sonore.

```
from pyo import *
s = Server().boot()
a = FM(carrier=[50,74.6,101,125.5], ratio=[1.501,2.02,1.005],
       index=[6,5,4,7,6,5,9,6,5,8,5,6], mul=.05).out()
print(len(a))
s.gui(locals())
```

scripts/chapitre_04/03_fm_stack.py

Les listes données aux arguments de l'objet *FM* étant de longueur différentes, la plus grande est utilisée pour déterminer le nombre de *streams* de l'objet, c'est-à-dire 12 (longueur de la liste donnée à l'argument « index »). En bouclant sur les valeurs des plus petites listes, nous obtenons 12 modulations de fréquence avec comme paramètres :

| **FM** | **carrier** | **ratio** | **index** |
|---|---|---|---|
| 1 | 50 | 1.501 | 6 |
| 2 | 74.6 | 2.02 | 5 |
| 3 | 101 | 1.005 | 4 |
| 4 | 125.5 | 1.501 | 7 |
| 5 | 50 | 2.02 | 6 |
| 6 | 74.6 | 1.005 | 5 |
| 7 | 101 | 1.501 | 9 |
| 8 | 125.5 | 2.02 | 6 |
| 9 | 50 | 1.005 | 5 |
| 10 | 74.6 | 1.501 | 8 |
| 11 | 101 | 2.02 | 5 |
| 12 | 125.5 | 1.005 | 6 |

Distribution des valeurs aux arguments des différentes modulation de fréquence.

**Mise en garde**

Un objet *pyo* étant considéré comme une liste, il est très facile de créer une surcharge de calcul pour le processeur en demandant de créer plusieurs instances d'un effet déjà couteux (une réverbération par exemple) sur un signal. La solution sera de passer un mix de tous les signaux (*streams*) de la source à l'effet. Prenons comme exemple notre lecture décalée (4 lectures stéréo) :

```
>>> a = SfPlayer("../snds/baseballmajeur_s.aif",
                 speed=[1,1.005,1.007,.992], loop=True, mul=.25)
>>> len(a)
8
>>> b = Freeverb(a).out()
>>> len(b)
8
```

CPU : 20%

```
>>> a = SfPlayer("../snds/baseballmajeur_s.aif",
                 speed=[1,1.005,1.007,.992], loop=True, mul=.25)
>>> len(a)
8
>>> b = Freeverb(a.mix(2)).out()
>>> len(b)
2
```

CPU : 16%

### 4.4.5 Exemples et exercices

**Exercices**

1. Écrire un script jouant 5 ondes sinusoïdales de fréquences croissantes {400, 500, 600, 700, 800} et d'amplitudes décroissantes {.5, .25, .12, .06, .03}. Les 5 ondes sinusoïdales doivent être générées avec une seule ligne de code !
2. Reprendre les exercices du cours précédents (chaînes de traitement à partir de sources de synthèse) et éliminer la méthode *ctrl()* en donnant des arguments à l'initialisation des objets. Générer plusieurs sonorités avec une même chaîne de traitement simplement en modifiant les paramètres d'initialisation (voir exemple 1 ci-dessous).
3. Magnifier les sons de l'exercice numéro 2 en utilisant le potentiel des listes en argument (voir exemple 2 ci-dessous).
4. Créer une chaîne de traitement sur une lecture de fichier son. Tester avec différents sons et différentes valeurs aux arguments des objets.

**Exemples**

Exemple numéro 1 : Spécification des arguments à l'initialisation des objets.

```
"""
Specification des arguments a l'initialisation des objets.

Trois sonorites sur la base de l'exemple no 3 du cours precedent :

Exemple no 3 : Connections en parallele (multi-sources) :
    lfo 1  ->
    lfo 2  ->  harmonizer
    lfo 3  ->

Exemple no 1 :
    Accord consonant avec une legere deviation des frequences
    pour creer un effet de modulation.
Exemple no 2 :
    Frequences des LFOs tres rapprochees et harmonisation de
    1/10e de demi-ton avec feedback pour creer un effet de flange.
Exemple no 3 :
    Desaccord total, sonorite se rapprochant du bruit.
"""
from pyo import *

s = Server().boot()

exemple = 1 # modifier cette variable pour changer d'exemple

if exemple == 1:
    src1 = LFO(freq=100, sharp=.5, mul=.1).out()
    src2 = LFO(freq=150.5, sharp=.25, mul=.1).out()
    src3 = LFO(freq=200.78, sharp=.15, mul=.1).out()
    harm = Harmonizer(src1+src2+src3, transpo=-7).out(1)
elif exemple == 2:
```

```
    src1 = LFO(freq=100, sharp=.75, mul=.1).out()
    src2 = LFO(freq=99.8, sharp=.75, mul=.1).out()
    src3 = LFO(freq=100.3, sharp=.75, mul=.1).out()
    harm = Harmonizer(src1+src2+src3, transpo=0.1, feedback=.8, mul=.6).out(1)
elif exemple == 3:
    src1 = LFO(freq=100, sharp=.75, mul=.1).out()
    src2 = LFO(freq=123, sharp=.65, mul=.1).out()
    src3 = LFO(freq=178, sharp=.5, mul=.1).out()
    harm = Harmonizer(src1+src2+src3, transpo=2.33, feedback=.5).out(1)

s.gui(locals())
```

scripts/chapitre_04/04_arguments.py

Exemple numéro 2 : Élargissement des sonorités à l'aide de listes en argument.

```
"""
Elargissement des sonorites a l'aide de listes en argument,
sur la base des trois sonorites de l'exemple precedent.

Exemple no 1 :
    Chorus sur chaque sources + multiples harmonisations.
Exemple no 2 :
    Frequences des LFOs tres rapprochees, sur 3 frequences
    et legeres harmonisations avec feedback pour creer un
    effet de flange.
Exemple no 3 :
    Desaccord total, encore plus proche du bruit.
"""
from pyo import *

s = Server().boot()

exemple = 1 # modifier cette variable pour changer d'exemple

if exemple == 1:
    s1 = LFO(freq=[100,100.03,99.95,99.91], sharp=.5, mul=.05).out()
    s2 = LFO(freq=[150.41,150.5,150.78,150.98], sharp=.25, mul=.05).out()
    s3 = LFO(freq=[200.78,201,201.3,202.1], sharp=.15, mul=.05).out()
    h = Harmonizer(s1+s2+s3, transpo=[-12,-7,5,7], mul=[1,.7,.5,.4]).out()
elif exemple == 2:
    s1 = LFO(freq=[100,300,500], sharp=.75, mul=[.1,.02,.015]).out()
    s2 = LFO(freq=[99.8, 300.3,500.5], sharp=.75, mul=[.1,.02,.015]).out()
    s3 = LFO(freq=[100.3,299.76,499,65], sharp=.75, mul=[.1,.02,.015]).out()
    h = Harmonizer(s1+s2+s3, transpo=[-.07,.04,.1], feedback=.8, mul=.4).out(1)
elif exemple == 3:
    s1 = LFO(freq=[100,276,421,511], sharp=.75, mul=[.1,.03,.02,.01]).out()
    s2 = LFO(freq=[123,324,389,488], sharp=.65, mul=[.1,.03,.02,.01]).out()
    s3 = LFO(freq=[178,265,444,561], sharp=.5, mul=[.1,.03,.02,.01]).out()
    h = Harmonizer(s1+s2+s3, transpo=[-3.76,2.33], feedback=.5).out(1)

s.gui(locals())
```

scripts/chapitre_04/05_list_in_args.py

# Chapitre 5

# Boucles et importation de modules

## 5.1 Importation de modules

Il est impossible d'importer automatiquement toutes les fonctions disponibles sous Python dès l'ouverture du logiciel, car il en existe virtuellement une infinité. Les fonctions intégrées représentent celles qui risquent d'être utilisées le plus souvent. Toutes les autres fonctions sont regroupées par thèmes dans des fichiers .py qui peuvent être importés en tant que **modules** (*pyo* est un module). Pour utiliser les fonctions contenues dans un module, il faut absolument importer ce module. Il existe deux formes d'importation sous Python.

#### Première forme : importer un module au complet

```
import sys, time
print(sys.path)
print(time.time())
```

L'attribut *path* du module **sys**, ainsi que la fonction *time* du module **time** sont alors accessibles. Comme c'est le module lui-même qui a été importé, il faut appeler l'attribut ou la fonction en spécifiant d'abord le nom du module (*module.attribut* ou *module.fonction()*).

#### Deuxième forme : importer des éléments d'un module

```
from random import randint, uniform
print(randint(100,200))
print(uniform(0,1))
```

Ici, seulement les fonctions *randint* et *uniform* du module **random** ont été importées. Elles peuvent donc être utilisées sans spécifier le nom du module auquel elles appartiennent.

#### Importer toutes les fonctions d'un module

```
from random import *
```

L'étoile signifie 'tous' !

#### Modifier le nom de référence

On peut ausssi modifier le nom de référence d'un module (principalement pour alléger la syntaxe) avec le mot-clé **as**. Dans l'exemple ci-dessous, toutes les fonctions du module *wx.html* pourront être appelées avec la syntaxe *html.fonction()*.

```
import wx.html as html

win = html.HtmlWindow()
```

La fonction **dir**(*module*) renvoie la liste des attributs et des fonctions appartenant à un module.

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil',
'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp',
'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi', 'pow', 'radians', 'sin',
'sinh', 'sqrt', 'tan', 'tanh']
>>>
```

## 5.2 Instructions répétitives (while, for)

Nous avons déjà exploré 2 des 3 principaux types de structures de contrôle en programmation, la **séquence** et l'**instruction conditionnelle**. Nous allons maintenant élaborer sur le troisième type, la **répétition**. Deux méthodes pour programmer des boucles seront présentées ici, l'instruction **while**, permettant de créer des boucles ayant une durée indéterminée, et l'instruction **for**, permettant de naviguer de façon cyclique sur tous les éléments d'un groupe.

### 5.2.1 L'instruction while

Prenons un exemple simple :

```
a = 0
while a < 10:
    # bloc d'instructions
    print(a)
    a += 1
# suite du programme
```

Une variable *a* est d'abord initialisée. Ensuite l'instruction **while** est appelée suivie d'une condition (ne pas oublier les deux-points qui indiquent le début d'un bloc instructions). **while**, qui veut dire « tant que » , signifie que tant que la condition est respectée, le bloc d'instructions qui suit (notez l'indentation) doit être répété en boucle. Voyons dans le détail :

1. l'instruction **while** évalue la condition. Si elle est fausse, tout le bloc est ignoré et le programme saute à la ligne suivant le bloc. Si la condition est vraie, le programme exécute les instructions du bloc :
2. la valeur de la variable *a* est affichée.
3. la variable *a* est incrémentée de 1.
4. l'instruction **while** est ensuite réévaluée en fonction de la nouvelle valeur de la variable *a*. Si la condition est encore vrai, le bloc est exécuté à nouveau, jusqu'à ce que la condition s'avère fausse.

Il est donc très important, afin de ne pas créer de boucles infinies, qu'il y ait au moins une variable de la condition qui soit modifiée à l'intérieur du bloc d'instructions de la boucle. Sous certaines conditions, des boucles infinies peuvent être définies dans un programme. Dans ce cas, une action de l'utilisateur, ou une commande envoyée lorsque l'on quitte le programme, met fin à la boucle.

Si une variable est utilisée dans la condition d'une instruction **while**, elle doit obligatoirement être définie avant d'exécuter cette instruction.

Les valeurs booléennes **True** et **False** peuvent servir à créer une boucle infinie, ou de façon plus commune, à activer ou désactiver un bloc de code dans un programme.

```
while True:
    # La condition est toujours vrai...
if False:
    # Ces instructions ne seront jamais executees, remplacer par:
if True:
    # Reactive le bloc d'instructions
```

Création d'une table des carrés et des cubes des nombres de 1 à 10 :

```
>>> a = 0
>>> while a < 10:
...     a += 1
...     print(a, a**2, a**3)
...
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

Nous allons maintenant créer un programme permettant d'afficher les nombres de la suite de *Fibonacci* (où une valeur est la somme des 2 valeurs précédentes). L'affichage arrêtera lorsqu'un certain seuil (défini par l'utilisateur) sera atteint.

```
1 seuil = 100
2 a = b = 1
3 while b < seuil:
4     print(b, end=" ")
5     a, b = b, a+b
6 print()
```

scripts/chapitre_05/01_fibonacci.py

Ce programme retournera le résultat suivant :

```
1 2 3 5 8 13 21 34 55 89
```

Il y a plusieurs aspects intéressants dans ce petit programme :

1. Affectations multiples permettant de donner la valeur 1 aux variables *a* et *b* simultanément.
2. Notez l'argument *end* donné à la fonction **print**. Il élimine le retour de chariot inséré automatiquement par la fonction **print** (en insérant plutôt un espace), permettant ainsi d'afficher plusieurs valeurs sur une même ligne.
3. Affectations parallèles à l'intérieur de la boucle ( a, b = b, a+b ). D'un seul coup, on modifie l'état des deux variables (*a* et *b*).

### 5.2.2 L'instruction for

L'instruction **for** sert à naviguer de façon cyclique sur tous les membres d'un groupe, habituellement une liste. Chaque membre sera assigné, à tour de rôle, à une variable pouvant être utilisée à l'intérieur de la boucle. La syntaxe d'une boucle **for** est très simple :

```
>>> for i in range(5):
...     print(i, end=" ")
...
0 1 2 3 4
```

La fonction **range**, avec un seul argument, retourne un itérateur générant une liste contenant les valeurs de 0 à la valeur en argument - 1. Si deux arguments sont présents, ils représenteront la valeur de départ et la valeur d'arrivée (non comprise) de la liste. Un troisième argument peut être utilisé pour définir le « pas », c'est à dire la valeur de l'incrément entre chaque membre de la liste.

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> range((5, 10))
[5, 6, 7, 8, 9]
>>> range((0,10,2))
[0, 2, 4, 6, 8]
```

La boucle **for** est très utile pour appliquer une transformation sur toutes les valeurs d'une liste :

```
>>> list1 = [2,4,6,8,10]
>>> list2 = []
>>> for i in list1:
...     # 'i' prend la valeur de chaque element de la liste a tour de role
...     list2.append(i**2)
...
>>> print(list2)
[4,16,36,64,100]
```

La méthode **append** fait partie des méthodes de la classe *list* et permet d'ajouter une valeur à la fin de la liste sur laquelle est appliquée la méthode. On utilise une méthode à l'aide de la syntaxe *objet.méthode*().

Variante du programme précédant, en utilisant la longueur de la liste :

```
>>> list1 = [2,4,6,8,10]
>>> for i in range(len(list1)):
...     # 'i' est un index dans la liste, 0 -> len(list1) - 1
...     list1[i] = list1[i]**2
...
>>> print(list1)
[4,16,36,64,100]
```

**len** est une fonction intégrée de Python qui retourne le nombre d'éléments contenus dans un groupe (liste, tuple, dictionnaire, chaîne de caractères) donné en argument. Cette variante constitue ce que l'on appelle une action destructive sur un ensemble (*in-place modification*), les valeurs de la liste sont directement remplacées sans avoir recours à une seconde liste.

## 5.3 Générateurs de listes ('list comprehension')

La fabrication de listes est une des fonctionnalités les plus puissantes du langage Python. Elle permet de créer des listes complexes très rapidement par une utilisation particulière de la boucle **for**. Les 'list comprehension' sont délimitées par des crochets et consistent en une expression suivie d'une boucle **for**, avec possibilité d'imbriquer plusieurs boucles **for** ou expressions **if**.

```
[ 'expression' 'boucle' 'condition (optionel)' ]
```

- Les **crochets** indique la création d'une nouvelle liste.
- **expression** est le bloc d'instructions de la boucle **for** (habituellement une expression mathématique).
- **boucle** est l'instruction répétitive **for** permettant d'exécuter l'expression un certain nombre de fois.
- **condition** est un paramètre optionnel permettant de spécifier sous quelles conditions l'expression doit être exécutée.

### Générateur simple

```
>>> ma_list = [i for i in range(20)]
>>> ma_list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>>
>>> carres = [i*i for i in range(5)]
>>> carres
[0, 1, 4, 9, 16]
```

### Générateur avec boucles for imbriquées

```
>>> ma_list = [[i,j] for i in range(3) for j in range(3)]
>>> ma_list
[[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1], [2, 2]]
```

Dans cet exemple, pour chaque tour de boucle de la variable $i$, la boucle de la variable $j$ est exécutée trois fois. La version longue de la création de cette liste s'écrirait comme suit :

```
ma_liste = []
for i in range(3):
    for j in range(3):
        ma_liste.append([i,j])
print(ma_liste)
```

### Générateur avec une condition

Dans le cas d'un générateur avec condition, l'expression sera exécutée (et le résultat sera ajouté à la liste) seulement pour les tours de boucle où la condition est respectée.

```
>>> ma_list = [[i,j] for i in range(3) for j in range(3) if j != 1]
>>> ma_list
[[0, 0], [0, 2], [1, 0], [1, 2], [2, 0], [2, 2]]
```

Version longue :

```
ma_liste = []
for i in range(3):
    for j in range(3):
        if j != 1:
            ma_liste.append([i,j])
print(ma_liste)
```

**Temps d'exécution**

Un des grands avantages des générateurs de listes est l'optimisation du code sous-jacent qui améliore de façon notable la performance du programme. Petit exemple qui démontre le temps requis pour créer une liste contenant un million d'entiers :

```
import time

num = 1000000

# "list comprehension"
t = time.time()
l = [i for i in range(num)]
print(time.time() - t) # 0.193097114563 sec.

# boucle "for"
t = time.time()
l = []
for i in range(num):
    l.append(i)
print(time.time() - t) # 0.299580097198 sec.

# toujours utiliser une fonction "built-in" si disponible!
t = time.time()
l = list(range(num))
print(time.time() - t) # 0.0406420230865 sec.
```

## 5.4 Pratique

### 5.4.1 Utilisation de la boucle *for* pour la génération de paramètres

Quelques exemples d'utilisation de la boucle *for* pour faciliter la construction de listes de paramètres.

**Génération d'une synthèse additive par l'addition d'ondes sinusoïdales dont les fréquences sont des multiples entiers d'une fréquence fondamentale.**

Les incontournables :

```
from pyo import *
s = Server().boot()
```

Création de listes vides où seront ajoutées les valeurs assignées aux paramètres de fréquence et d'amplitude :

```
freqs = []
amps = []
```

On définit une fréquence fondamentale :

```
fond = 50
```

À l'intérieur d'une boucle *for*, nous allons remplir nos listes en appliquant de simples fonctions mathématiques sur la variable $i$ (qui prendra tour à tour les valeurs de 1 à 20) :

```
for i in range(1,21):
    freqs.append(i*fond)
    amps.append(0.35/i)
```

Affichage des listes remplies :

```
print('Frequences: ', freqs)
print('Amplitudes: ', amps)
```

En donnant nos listes aux paramètres d'un objet *Sine*, nous créons autant d'oscillateurs qu'il y a de valeurs dans les listes :

```
a = Sine(freq=freqs, mul=amps).out()
```

Autre incontournable :

```
s.gui(locals())
```

**Questions :**

- Que doit-on faire pour obtenir une fréquence fondamentale différente ?
- Que doit-on faire pour ajouter ou enlever des harmoniques ?

```
1  # Exercice boucle "for", serie de frequences et d'amplitudes.
2  from pyo import *
3
4  s = Server().boot()
5
6  freqs = []
7  amps = []
8
9  fond = 50
10 for i in range(1,21):
11     freqs.append(i*fond)
12     amps.append(0.35/i)
13
14 print('Frequences : ', freqs)
15 print('Amplitudes : ', amps)
16
17 a = Sine(freq=freqs, mul=amps).out()
18
19 s.gui(locals())
```

scripts/chapitre_05/02_boucle_for_1.py

**Modification de l'exemple précédant afin de lui donner un peu plus de naturel...**

Le principal défaut de ce type d'algorithme est la trop grande précision des valeurs. Aucun son naturel ne peut produire des harmoniques aussi justes et c'est justement dans les infimes variations que le son prend vie. Il faut donc trouver un moyen de désaccorder les harmoniques les unes par rapport aux autres. Pour ce faire, nous allons nous en remettre au "hasard" ainsi qu'au module python *random*, générateur de hasard par excellence !

Commençons par importer le module *random* :

```
import random
```

Nous allons ensuite modifier notre boucle *for* afin d'inclure une légère variation de la fréquence de chaque harmonique :

```
    dev = random.uniform(0.98, 1.02)
    freqs.append(i*fond*dev)
```

La fonction *uniform* du module *random* demande 2 paramètres qui sont respectivement la valeur minimum possible et la valeur maximum possible. Chaque fois que la fonction est appelée, une nouvelle valeur est pigée dans l'ambitus permis. On multiplie la fréquence de l'oscillateur par cette valeur pour créer une légère déviation.

Testez différents ambitus de déviation afin de constater l'impact sur le son !

```
"""
Exercice boucle "for", version 2, serie de frequences et d'amplitudes.

"""
from pyo import *
import random

s = Server().boot()

freqs = []
amps = []

fond = 50
for i in range(1,21):
    dev = random.uniform(.98,1.02)
    freqs.append(i*fond*dev)
    amps.append(0.35/i)

print('Frequences : ', freqs)
print('Amplitudes : ', amps)

a = Sine(freq=freqs, mul=amps).out()

s.gui(locals())
```

scripts/chapitre_05/03_boucle_for_2.py

**Synthèse par modulation de fréquence en utilisant quelques une des fonctions offertes par le module random.**

Le script suivant utilise des générateurs de distributions aléatoires contrôlées, accessibles via le module *random*, pour créer une sonorité de synthèse par modulation de fréquence différente chaque fois que le script est exécuté.

— **random.triangular(min, max)** : Distribution triangulaire (favorise les valeurs médianes) entre *min* et *max*.
— **random.choice(seq)** : Pige une valeur au hasard dans la liste donnée au paramètre *seq*.
— **random.randint(min, max)** : Pige un entier au hasard entre *min* et *max*.

Consultez le manuel Python pour une description détaillée des fonctions du module random.

```
"""
Exercice boucle "for", synthese par modulation de frequence.

"""
from pyo import *
import random

s = Server().boot()

car = []
rat = []
ind = []
for i in range(10):
    car.append(random.triangular(150, 155))
    rat.append(random.choice([.25, .5, 1, 1.25, 1.5, 2]))
    ind.append(random.randint(2, 6))

print('Carrier : ', car)
print('Ratio : ', rat)
print('Index : ', ind)

fm = FM(carrier=car, ratio=rat, index=ind, mul=.05).out()

s.gui(locals())
```

scripts/chapitre_05/04_boucle_for_FM.py

### 5.4.2 Les générateurs de listes ("list comprehension")

Parfois, les opérations nécessaires à la génération d'une liste ne nécessite pas plusieurs lignes de code. Il est alors possible d'insérer l'expression à même un générateur de liste. Cette technique permet par exemple de créer un immense chorus en une seule ligne :

```
a = FM([random.uniform(240, 260) for i in range(200)], mul=.005).out()
```

200 synthèses FM ont été créés par cette ligne.

On pourrait reproduire l'exemple de synthèse par modulation de fréquence à l'aide des générateurs de listes :

```
from pyo import *
import random

s = Server().boot()

fm = FM(carrier=[random.triangular(150, 155) for i in range(10)],
        ratio=[random.choice([.25, .5, 1, 1.25, 1.5, 2]) for i in range(10)],
        index=[random.randint(2, 6) for i in range(10)], mul=.05).out()

s.gui(locals())
```

scripts/chapitre_05/05_listComp_FM.py

**Exercices**

**1** - Dans le script suivant, vous devez remplacer les 3 lignes vides (...) de façon à ce que votre script génère un chorus de 10 voix (vitesses légèrement différentes) par la lecture d'un son en boucle sur le disque dur. Vous devez créer un mix mono à partir du chorus et ensuite lui appliquer une harmonisation à 3 voix comportant l'octave inférieur (-12), la tierce supérieure (4) et la quinte supérieure (7). Faites attention aux amplitudes ! La répartition des voix se fait comme suit :

À gauche : Le chorus original et la quinte supérieure.
À droite : L'octave inférieur et la tierce supérieure.

```
from pyo import *
from random import uniform
s = Server().boot()

...
...
...

s.gui(locals())
```

**2.1** - Sachant qu'une onde carrée n'est composée que d'harmoniques impairs dont les amplitudes correspondent à l'inverse de leur rang, quels seraient les générateurs de listes qui nous permettraient d'obtenir les fréquences et les amplitudes correspondantes d'un tel son ?

Pour une fondamentale de 100 Hz, on veut les composantes suivantes :

100 Hz, amplitude = 1
300 Hz, amplitude = 1/3.
500 Hz, amplitude = 1/5.
700 Hz, amplitude = 1/7.
...

**2.2** - Transformez vos générateurs de listes afin de créer un chorus de cette note... !

Les solutions en page 68.

### 5.4.3 Variations continues des paramètres à l'aide d'objets *pyo*

Un son naturel n'est pas statique, il est généralement en constante évolution. Que ce soit par de légères variations de fréquence ou d'amplitude, le timbre d'un son varie au cours du temps. Nous allons maintenant commencer à utiliser des objets *pyo* dans le but de créer des trajectoires de contrôle, c'est-à-dire des signaux audio non pas destinés à l'envoi vers les haut-parleurs mais plutôt à créer des variations de paramètres dans le temps. L'utilisation en est fort simple. Si le manuel spécifie qu'un argument accepte les "floats" **ou** les PyoObjects, alors il est permis de lui donner un objet *pyo* préalablement créé. Pour donner une légère variation de fréquence à un oscillateur, le script suivant crée un objet qui génère des valeurs aléatoires continues (*Randi*) que l'on assigne à l'argument *freq* d'un oscillateur :

```
from pyo import *
s = Server().boot()

rnd = Randi(min=390, max=410, freq=4)
a = Sine(freq=rnd, mul=.5).out()

s.gui(locals())
```

scripts/chapitre_05/09_randi.py

4 fois par seconde, *Randi* pige une nouvelle valeur entre 390 et 410 et interpole linéairement de la valeur courante à la valeur pigée.

#### Génération d'une note *chorussée*

Il faut se rappeler qu'un objet *pyo* est considéré comme une liste (la longueur de la liste correspond au nombre de *streams* audio qu'il contient). Étant considéré comme une liste, si on assigne à un argument un objet audio d'une longueur $x$, l'objet nouvellement créé contiendra lui aussi $x$ *streams* audio. C'est une arme à double tranchant ! Il est très facile de saturer le processeur par une démultiplication des calculs mais il est aussi très aisé de construire des sonorités complexes en quelques lignes. Voici un exemple d'une note *chorussée* :

```
from pyo import *
import random

s = Server().boot()

fr = Randi(min=295, max=300, freq=[random.uniform(2,8) for i in range(100)])
sines = SineLoop(fr, feedback=.08, mul=.01).out()
print(len(sines))

s.gui(locals())
```

scripts/chapitre_05/10_var_freqs.py

L'objet *Randi*, en recevant une liste de 100 valeurs à l'argument *freq*, créera 100 variations aléatoires. En assignant cet objet au paramètre *freq* d'un oscillateur, celui-ci génèrera automatiquement 100 oscillateurs de fréquences différentes. Prenez garde à l'amplitude !

Un exemple similaire, cette fois-ci les variations sont appliquées à l'amplitude de 10 oscillateurs :

```
from pyo import *
import random

s = Server().boot()

amp = Randi(min=0, max=.05, freq=[random.uniform(.25,1) for i in range(10)])
sines = SineLoop(freq=[i*100+50 for i in range(10)], feedback=.03, mul=amp).out()

s.gui(locals())
```

scripts/chapitre_05/11_var_amps.py

### Exercices

1. Pour les deux exemples précédents, testez différentes valeurs aux arguments *min*, *max* et *freq* des objets *Randi* afin de figurer l'impact de chacun sur le résultat sonore.
2. Remplacez les objets *Randi* par des objets *Randh*, quelle est la différence ?

### Génération de nombres aléatoires prédéterminés

Il est parfois désirable de pouvoir prédire, dans une certaine mesure, la teneur d'une génération aléatoire. La génération d'une harmonie est un cas de figure où les valeurs de fréquence ne peuvent pas être simplement soumises au hasard. L'objet *Choice* nous permet de spécifier, à l'aide d'une liste, les valeurs qui pourront sortir de la pige. Voici un petit exemple, à 4 voix de polyphonie, où les fréquences sont restreintes à des notes de la gamme de Do majeur :

```
from pyo import *

s = Server().boot()

pits = [midiToHz(m) for m in [36,43,48,55,60,62,64,65,67,69,71,72]]
choix = Choice(choice=pits, freq=[1,2,3,4])
sines = SineLoop(freq=choix, feedback=.05, mul=.1).out()

s.gui(locals())
```

scripts/chapitre_05/12_carillon.py

La fonction *midiToHz* convertit une valeur de note Midi (un entier entre 0 et 127) en la fréquence correspondante en Hertz.

### 5.4.4 Exemples et exercices

#### Exercices

1. Reprendre les exercices précédents et remplacer les sources sonores pour tester différentes sonorités.
2. Appliquer des générateurs aléatoires sur les différents paramètres de contrôle des sources. Explorer les objets de la catégories *randoms* de pyo.
   * Attention * C'est à partir d'ici que c'est dangereux pour vos oreilles ! Vérifiez à bas volume si vos automations ne provoquent pas des écarts dynamiques trop grands.
3. Compléter les chaînes de traitement par l'ajout d'effets. Les catégories *filters*, *effects* et *dynamics* offrent plusieurs processus de transformation du signal.

#### Exemples

L'exemple ci-dessous se base sur les 4 voix de polyphonie, avec pige de fréquences aléatoires sur la gamme de Do, du script précédent.

```
pits = [midiToHz(m) for m in [36,43,48,55,60,62,64,65,67,69,71,72]]
choix = Choice(choice=pits, freq=[1,2,3,4])
sines = SineLoop(freq=choix, feedback=.05, mul=.1).out()
```

Dans un premier temps, doublons la source à l'aide d'un générateur de synthèse par modulation de fréquence (**FM**). En prime, un petit portamento a été ajouté aux changements de fréquences afin d'éliminer les clics potentiels.

```
pits = [midiToHz(m) for m in [36,43,48,55,60,62,64,65,67,69,71,72]]
choix = Choice(choice=pits, freq=[1,2,3,4])
# Petit truc, un portamento sur les changements de frequence elimine les clics!
ch_port = Port(choix, risetime=.001, falltime=.001)
sines = SineLoop(freq=ch_port, feedback=.05, mul=.1).out()
# Seconde source, synthese FM
fms = FM(carrier=ch_port, ratio=1.0025, index=4, mul=.025).out()
```

Ajoutons deux automations sur le *feedback* du *SineLoop* et sur l'*index* de la modulation de fréquence. Les deux LFO sont en inversion de phase afin d'alterner la prédominance d'une source par rapport à l'autre.

```
pits = [midiToHz(m) for m in [36,43,48,55,60,62,64,65,67,69,71,72]]
choix = Choice(choice=pits, freq=[1,2,3,4])
ch_port = Port(choix, risetime=.001, falltime=.001)
# LFO, en phase, sur le feedback du SineLoop
lffeed = Sine(freq=0.1, mul=.07, add=.07)
sines = SineLoop(freq=ch_port, feedback=lffeed, mul=.1).out()
# LFO, hors phase, sur l'index de la FM
lfind = Sine(freq=0.1, phase=0.5, mul=3, add=3)
fms = FM(carrier=ch_port, ratio=1.0025, index=lfind, mul=.025).out()
```

Ajoutons maintenant un premier effet, un effet de "flange", créé à l'aide du ligne de délai dont le temps de délai varie dans le temps. Tout d'abord, il ne faut pas oublier de mixer les *streams* audio en stéréo si nécessaire (nous avons 4 voix de polyphonie) et de sommer les sources. Puis nous ajoutons le délai avec un LFO sinusoïdal sur le paramètre *delay*, ce qui aura pour effet un balayage du spectre par un filtre en peigne. Notez que les méthodes *out()* ont été retirées des objets *SineLoop* et *FM* pour n'agir que sur la somme des deux sources.

```
pits = [midiToHz(m) for m in [36,43,48,55,60,62,64,65,67,69,71,72]]
choix = Choice(choice=pits, freq=[1,2,3,4])
ch_port = Port(choix, risetime=.001, falltime=.001)
lffeed = Sine(freq=0.1, mul=.07, add=.07)
sines = SineLoop(freq=ch_port, feedback=lffeed, mul=.1)
lfind = Sine(freq=0.1, phase=0.5, mul=3, add=3)
fms = FM(carrier=ch_port, ratio=1.0025, index=lfind, mul=.025)
# Addition du mix stereo des deux sources
src_sum = sines.mix(2) + fms.mix(2)
# Delai avec LFO sur le temps de delai pour creer un effet de flange
lfdel = Sine(.1, mul=.003, add=.005)
comb = Delay(src_sum, delay=lfdel, feedback=.5).out()
# Envoie la somme aux haut-parleurs
src_sum.out()
```

Finalement, on dirige le signal vers une réverbération qui assumera la balance entre le signal original et le signal réverbéré ainsi que l'envoie au haut-parleurs. Voici la version finale de notre programme :

```
1  from pyo import *
2
3  s = Server().boot()
4
5  pits = [midiToHz(m) for m in [36,43,48,55,60,62,64,65,67,69,71,72]]
6
7  choix = Choice(choice=pits, freq=[1,2,3,4])
8  ch_port = Port(choix, risetime=.001, falltime=.001)
9
10 lffeed = Sine(freq=0.1, mul=.07, add=.07)
11 sines = SineLoop(freq=ch_port, feedback=lffeed, mul=.1)
12
13 lfind = Sine(freq=0.1, phase=0.5, mul=3, add=3)
14 fms = FM(carrier=ch_port, ratio=1.0025, index=lfind, mul=.025)
15
16 src_sum = sines.mix(2) + fms.mix(2)
17
18 lfdel = Sine(.1, mul=.003, add=.005)
19 comb = Delay(src_sum, delay=lfdel, feedback=.5)
20
21 out_sum = src_sum + comb # Sommation des sources et du delai
22 # Envoie vers le reverbe et sortie du signal
23 rev = WGVerb(out_sum, feedback=.8, cutoff=3500, bal=.4).out()
24
25 s.gui(locals())
```

scripts/chapitre_05/13_exemple_final.py

### 5.4.5 Solutions aux exercices sur les 'lists comprehension'

```
#!/usr/bin/env python
# encoding: utf-8
"""
Solution pour l'exercice 1 sur les 'lists comprehension'.

"""
from pyo import *
from random import uniform

s = Server().boot()

sf = SfPlayer(SNDS_PATH+'/transparent.aif', loop=True, mul=.07,
              speed=[uniform(.99,1.01) for i in range(10)]).mix(1)
sf.out()
harm = Harmonizer(sf, transpo=[-12,7,4], mul=.4).out(1)

s.gui(locals())
```

scripts/chapitre_05/06_ex_1_listComp.py

```
#!/usr/bin/env python
# encoding: utf-8
"""
Solution pour l'exercice 2.1 sur les 'lists comprehension'.

"""
from pyo import *
from random import uniform

s = Server().boot()

max_harms = 20
freqs = [i * 100 for i in range(max_harms) if (i%2) == 1]
amps = [0.35 / i for i in range(max_harms) if (i%2) == 1]
print("Frequences :", freqs)
print("Amplitudes :", amps)
a = Sine(freq=freqs, mul=amps).mix(1).out()

s.gui(locals())
```

scripts/chapitre_05/07_ex_2_1_listComp.py

```
#!/usr/bin/env python
# encoding: utf-8
"""
Solution pour l'exercice 2.2 sur les 'lists comprehension'.

"""
from pyo import *
from random import uniform

s = Server().boot()

max_harms = 20

freqs = [i * 100 * uniform(.99,1.01)
         for i in range(max_harms) for j in range(10) if (i%2) == 1]

amps = [0.035 / i for i in range(max_harms) for j in range(10) if (i%2) == 1]

print("Frequences :", freqs)
print("Amplitudes :", amps)

a = Sine(freq=freqs, mul=amps).mix(1).out()

s.gui(locals())
```

scripts/chapitre_05/08_ex_2_2_listComp.py

# Chapitre 6

# Les fonctions

Ce cours élabore sur la création de fonctions personnalisées et l'utilisation de celles-ci à l'intérieur d'un script.

## 6.1 Fonction simple sans paramètre

Créer une fonction consiste à regrouper un certain nombres d'opérations, reliées entre elles, effectuant un processus particulier. Une fonction préalablement définie peut-être appelée n'importe où dans un script, permettant une écriture compacte et efficace des programmes. Le groupe d'opérations peut ne comporter que quelques lignes ou être un regroupement complexe d'opérations s'étendant sur plusieurs centaines de lignes. Voici un exemple simple de fonction sans paramètre et son utilisation dans un script :

```
from math import sqrt

def racine_carree_2_4():
    print('racine carree de 2 : %f' % sqrt(2))
    print('racine carree de 3 : %f' % sqrt(3))
    print('racine carree de 4 : %f' % sqrt(4))

racine_carree_2_4()
```

Noter la création d'un bloc d'instructions avec les deux-points terminant la ligne d'en-tête et l'indentation du bloc.

Cette fonction ne fait qu'imprimer les racines carrées des nombres 2 à 4, mais elle illustre déjà la syntaxe de base nécessaire à la création d'une fonction. Celle-ci commence toujours par le mot-clé **def** suivit du nom de la fonction et de parenthèses (nous verrons plus loin qu'elles ne sont pas toujours vides). Les deux-points terminent la ligne d'en-tête et toutes les lignes indentées qui suivent constituent le bloc d'instructions qui sera exécuté chaque fois que la fonction sera appelée. Pour terminer la construction de la fonction, il suffit de revenir à la ligne d'indentation principale.

## 6.2 Fonction avec paramètres

Lors de la création d'une fonction, il est possible de spécifier des paramètres en leur attribuant un argument entre les parenthèses. Les arguments prendront les valeurs données lors de l'appel de la fonction et pourront être utilisés comme des variables normales dans l'exécution du bloc d'instructions. Ceci permet d'appliquer une suite d'opérations sur différentes valeurs simplement en variant les paramètres donnés à l'appel.

```
from math import sqrt

def racine_carree(x):
    print('racine carree de %f : %f' % (x, sqrt(x)))

racine_carree(25)
racine_carree(12)
```

Si plus d'un paramètre sont nécessaires au bon déroulement de la fonction, il suffit de les séparer par une virgule entre les parenthèses.

```
from math import sqrt

def racines_carre(mini, maxi):
    print('les racines carrees des nombre entre %d et %d sont :' % (mini, maxi))
    for i in range(mini, maxi):
        print(sqrt(i), end=" ")

racines_carre(5, 10)
```

## 6.3 Valeurs par défaut des paramètres

Une fonctionnalité particulièrement intéressante consiste à donner des valeurs par défaut aux paramètres d'une fonction. Ceci permet d'omettre certains paramètres lors de l'appel sans créer d'erreurs d'exécution. Une fonction peut donc être créée en donnant beaucoup de latitude à l'utilisateur tout en lui permettant de l'utiliser dans des variantes plus simples. À la création de la fonction, on attribue une valeur par défaut à un paramètre en faisant suivre l'argument correspondant du signe "=" puis de la valeur désirée.

```
def puissance(x, exp=2):
    print('%.2f exposant %.2f donne : %.2f' % (x, exp, pow(x,exp)))

puissance(5)
puissance(2, 16)
puissance(2, exp=32)
puissance(x=3, exp=2)
puissance(exp=3, x=2)
```

On constate qu'il y a plusieurs façons différentes d'appeler cette fonction. On peut spécifier seulement le paramètre sans valeur par défaut, ou bien spécifier les deux valeurs (la valeur donnée à l'argument *exp* remplace la valeur par défaut). Il est aussi possible de faire directement référence à la variable dont on veut spécifier la valeur avec la syntaxe *nom_de_la_variable=valeur*.

**Quelques règles à respecter**

1. Lors de la définition d'une fonction, les paramètres sans valeur par défaut doivent précéder les paramètres avec valeur par défaut.
2. Lors de l'appel, si on ne fait pas référence aux noms des variables, les valeurs doivent être spécifiées dans l'ordre.
3. Lors de l'appel, on ne peut définir un paramètre en spécifiant le nom de la variable et ensuite attribuer une valeur à un paramètre sans spécifier explicitement son nom. Lorsqu'un nom de variable est donné, l'ordre des paramètres n'est plus respecté, alors l'interpréteur ne saura pas à quel paramètre attribuer la valeur.

## 6.4 Variables locales et variables globales

À ce moment-ci, il est important de comprendre comment se partage les espaces de noms en Python. Les espaces de noms sont des registres où sont enregistrées les variables spécifiques à certains environnements, par exemple les différentes fonctions et classes. Afin d'éviter les conflits entre des variables portant le même nom, l'interpréteur parcours les espaces de noms dans un ordre précis pour identifier les différentes variables. En règle générale, l'interpréteur recherche dans l'espace le plus local possible, et s'il ne trouve pas la variable demandée, il remonte les étages jusqu'à l'espace de noms global. Nous aurons l'occasion d'y revenir et d'approfondir un peu plus ce sujet lorsque nous parlerons des classes et des objets. Pour l'instant, deux types de variables nous intéressent plus particulièrement, les variables locales et les variables globales.

Une variable globale est une variable définie au premier niveau d'indentation d'un script, donc accessible n'importe où dans le script, tandis qu'une variable locale est une variable définie à l'intérieur d'une fonction et n'est accessible que dans la fonction elle-même. Il est donc tout à fait possible qu'une variable initialisée au début d'un script porte un nom $x$, et qu'une variable portant le même nom soit définie à l'intérieur d'une fonction, et ce, sans créer de conflit. Il faut cependant avoir conscience de l'espace de noms où l'on se trouve afin de savoir quelle est la valeur réelle de la variable que l'on utilise. Petit exemple :

```
>>> a = 1
>>> def f():
...     a = 2
...
>>> f()
>>> print(a)
1
```

Dans cet exemple, une variable $a$ est initialisée à la valeur 1, puis une fonction, donnant la valeur 2 à une variable $a$, est appelée. Ensuite, la valeur de la variable $a$ est affichée et on

constate qu'elle est de 1, et non de 2, même si la dernière opération effectuée sur une variable $a$ est une affectation de la valeur 2. La raison est que ces deux variables ne font pas partie du même espace de noms, la première fait partie de l'espace de noms global du script, tandis que la deuxième fait partie de l'espace local de la fonction. Ce sont donc deux variables distinctes.

```
>>> a = 1
>>> def f():
...     a = 2
...     print(a)
...
>>> f()
2
>>> print(a)
1
```

Bien que la variable définie dans la fonction ne soit pas accessible de l'extérieur, le contraire est tout à fait possible. C'est à dire qu'il est permis d'accéder à la valeur d'une variable faisant partie de l'espace de noms global à l'intérieur d'une fonction, sans pouvoir la modifier... pour l'instant.

```
>>> a = 1
>>> def f():
...     print(a)
...
>>> f()
1
```

Si l'on désire agir sur le contenu d'une variable globale à l'intérieur d'une fonction, il faut la déclarer 'variable globale' à l'intérieur même de la fonction. Cette déclaration s'opère avec le mot-clé **global** et indique à l'interpréteur de ne pas créer une nouvelle variable locale mais d'utiliser, et de modifier, la variable globale.

```
>>> a = 1
>>> def f():
...     global a
...     a = 2
...
>>> f()
>>> print(a)
2
```

## 6.5 Utilisation des fonctions avec retour de valeur

L'utilité principale d'une fonction est d'effectuer des opérations sur des paramètres et de fournir un résultat. Ce résultat peut être retourné par la fonction et placé en mémoire dans une autre variable, pouvant être utilisée telle quelle dans la suite du programme. Lorsque le mot-clé **return** est rencontré dans une fonction, celui-ci termine la fonction et retourne la ou les valeurs qui le suivent, s'il y en a. On peut affecter, avec le symbole "=", le retour d'une fonction à une variable. Cette opération est illustrée dans l'extrait suivant.

```
>>> a = 0
>>> def increment(x):
...     return x + 1
...
>>> a = increment(a)
>>> print(a)
1
>>> a = increment(a)
>>> print(a)
2
```

Le retour d'une fonction peut être une valeur de n'importe quel type, par exemple, une liste :

```
a = 16

def liste_puissance_2(exp):
    exposants = []
    for i in range(1, exp+1):
        exposants.append(i**2)
    return exposants

maListe = liste_puissance_2(a)
```

## 6.6 Pratique

### 6.6.1 Utilisation des fonctions dans un script

Nous allons maintenant explorer l'interaction avec le signal audio d'un script à l'aide de fonctions préalablement définies. Comme premier exemple, nous allons définir 3 sources sonores (un son, une synthèse FM et une synthèse additive) qui seront activées à tour de rôle par l'appel d'une fonction. Les trois sources :

```
1  from pyo import *
2  import random
3  s = Server().boot()
4
5  # lecture d'un fichier son
6  snd = SfPlayer(SNDS_PATH + "/accord.aif", speed=[.998,1.003],
7                 loop=True, mul=.35).stop()
8  # synthese FM
9  fm = FM(carrier=[99,99.7,100,100.4,100.9], ratio=.4987, index=8, mul=.1).stop()
10 # synthese additive
11 adsyn = SineLoop(freq=[random.uniform(145,155) for i in range(20)],
12                  feedback=.15, mul=.05).stop()
13
14 s.gui(locals())
```

scripts/chapitre_06/01_interaction_1.py

Notez l'appel de la méthode *stop* sur chacune des sources. Ceci aura pour effet de ne pas lancer le calcul des échantillons automatiquement (la méthode *play* est presque toujours appelée dès la création d'un objet).

### Utilisation de l'entrée-texte dans la fenêtre du serveur

Maintenant que nous avons connaissance des espaces de noms de Python, nous pouvons éclaircir l'argument de la méthode *gui* de l'objet *Server*. *locals* est une fonction intégrée de Python qui retourne toutes les variables jusqu'alors définies dans l'espace de noms courant (l'espace de noms du script dans la plupart des cas). Ainsi, toutes les variables définies avant l'appel de la fenêtre du serveur sont connues de l'entrée-texte, qui est ni plus ni moins qu'une extension de l'interpréteur. Il est donc possible de lancer de nouvelles commandes à partir de cet outils. Cela servira à modifier l'état de certains objets ou à lancer des fonctions préalablement définies. Par exemple, pour démarrer le calcul de la synthèse FM, on écrira, suivi de la touche « enter » :

```
fm.out()
```

Ajoutons une fonction à notre script permettant de démarrer une source tout en arrêtant celle qui jouait précédemment. Un argument servira à déterminer quelle source doit jouer. Cette fonction s'insère après la définition des sources et avant l'affichage de la fenêtre du serveur.

```
from pyo import *
import random
s = Server().boot()

# lecture d'un fichier son
snd = SfPlayer(SNDS_PATH + "/accord.aif", speed=[.998,1.003],
               loop=True, mul=.35).stop()
# synthese FM
fm = FM(carrier=[99,99.7,100,100.4,100.9], ratio=.4987, index=8, mul=.1).stop()
# synthese additive
adsyn = SineLoop(freq=[random.uniform(145,155) for i in range(20)],
                 feedback=.15, mul=.05).stop()

# src -> {'snd', 'fm', 'adsyn'}
def play(src="snd"):
    if src == 'snd':
        snd.out()
        fm.stop()
        adsyn.stop()
    elif src == 'fm':
        snd.stop()
        fm.out()
        adsyn.stop()
    elif src == 'adsyn':
        snd.stop()
        fm.stop()
        adsyn.out()

```

```
29 s.gui(locals())
```

scripts/chapitre_06/02_interaction_2.py

Afin d'adoucir les entrées et sorties de son dans notre programme, nous allons appliquer une enveloppe d'amplitude à chacune de nos sources. L'objet *Fader* permet de définir un temps de montée (déclenché par la méthode *play*) et un temps de chute (déclenché par la méthode *stop*) d'un contrôle d'amplitude qui sera assigné à l'argument *mul* de la source. Nul besoin d'appeler la méthode *stop* sur les objets *Fader*, c'est un des rares cas où la méthode *play* n'est pas appelée automatiquement à la création de l'objet. Cette fois-ci, nous laisserons les sources jouer en permanence et ne contrôlerons que les amplitudes via la fonction de contrôle.

```
1  from pyo import *
2  import random
3  s = Server().boot()
4
5  snd_amp = Fader(fadein=5, fadeout=5, dur=0)
6  snd = SfPlayer(SNDS_PATH + "/accord.aif", speed=[.998,1.003],
7                 loop=True, mul=snd_amp*.35).out()
8  fm_amp = Fader(fadein=5, fadeout=5, dur=0)
9  fm = FM(carrier=[99,99.7,100,100.4,100.9], ratio=.4987, index=8,
10         mul=fm_amp*.1).out()
11 adsyn_amp = Fader(fadein=5, fadeout=5, dur=0)
12 adsyn = SineLoop(freq=[random.uniform(145,155) for i in range(20)],
13                  feedback=.15, mul=adsyn_amp*.05).out()
14
15 # src -> {'snd', 'fm', 'adsyn'}
16 def play(src):
17     if src == 'snd':
18         snd_amp.play()
19         fm_amp.stop()
20         adsyn_amp.stop()
21     elif src == 'fm':
22         snd_amp.stop()
23         fm_amp.play()
24         adsyn_amp.stop()
25     elif src == 'adsyn':
26         snd_amp.stop()
27         fm_amp.stop()
28         adsyn_amp.play()
29
30 s.gui(locals())
```

scripts/chapitre_06/03_interaction_3.py

### 6.6.2 Appel automatique d'une fonction avec l'objet *Pattern*

Le processus de l'objet *Pattern* est d'appeler une fonction de façon cyclique. L'argument *function* prend le nom d'une fonction préalablement définie (notez bien qu'il n'y a pas de parenthèses, c'est une référence et non un appel) et l'argument *time*, le temps écoulé entre chaque appel. Voici comment nous allons créer cet objet :

```
pat = Pattern(function=play, time=6).play()
```

La fonction *play* sera appelée à toutes les 6 secondes, sans paramètre. Nous allons donc devoir modifier cette dernière afin d'éliminer le paramètre qui pointe sur une source. Nous allons plutôt créer un mécanisme qui pige une source au hasard :

```
last = None
def play():
    global last
    src = random.choice(['snd', 'fm', 'adsyn'])
    while src == last:
        src = random.choice(['snd', 'fm', 'adsyn'])
    last = src
    if src == 'snd':
        ....
```

Plusieurs éléments à noter ici. Premièrement, après avoir choisi un *string* au hasard dans la liste des choix possibles (n'oubliez pas d'importer le module *random*), on pose une condition dans une boucle *while* pour s'assurer que la pige n'est pas identique à la précédente. La boucle exécutera une nouvelle pige jusqu'à ce que *src* ne soit pas égal à *last*. Ensuite, on remplace le contenu de la variable globale *last* par la nouvelle pige pour préparer la prochaine condition.

```
from pyo import *
import random
s = Server().boot()

snd_amp = Fader(fadein=5, fadeout=5, dur=0)
snd = SfPlayer(SNDS_PATH + "/accord.aif", speed=[.998,1.003],
               loop=True, mul=snd_amp*.35).out()
fm_amp = Fader(fadein=5, fadeout=5, dur=0)
fm = FM(carrier=[80,79.7,81,81.4,81.9], ratio=.4987, index=8, mul=fm_amp*.1).out()
adsyn_amp = Fader(fadein=5, fadeout=5, dur=0)
adsyn = SineLoop(freq=[random.uniform(145,155) for i in range(20)],
                 feedback=.15, mul=adsyn_amp*.05).out()

last = None
def play():
    global last
    src = random.choice(['snd', 'fm', 'adsyn'])
    while src == last:
        print('"%s" is already playing, choosing another...' % src)
        src = random.choice(['snd', 'fm', 'adsyn'])
    last = src
    print(src)
    if src == 'snd':
        snd_amp.play()
        fm_amp.stop()
        adsyn_amp.stop()
    elif src == 'fm':
        snd_amp.stop()
        fm_amp.play()
        adsyn_amp.stop()
    elif src == 'adsyn':
```

```
32          snd_amp.stop()
33          fm_amp.stop()
34          adsyn_amp.play()
35
36  pat = Pattern(function=play, time=6).play()
37  s.gui(locals())
```

scripts/chapitre_06/04_interaction_4.py

Les variables définies à l'intérieur d'une fonction sont automatiquement détruites lorsque l'exécution de la fonction est terminée, il est donc nécessaire de déclarer la variable *last* au premier niveau d'indentation du script. De cette façon, elle sera disponible pour toute la durée de vie du programme. Par contre, comme nous voulons la modifier à l'intérieur de la fonction, nous devons la déclarer comme étant une variable globale. Si on omet la ligne *global last*, une nouvelle variable locale à la fonction sera créée et ne sera plus disponible au prochain appel.

### 6.6.3 Génération d'une séquence composée (objet *Score*)

L'objet *Score* permet d'organiser le déroulement d'une suite d'événements dans le temps. L'argument *input* attend un signal audio, contenant des entiers, qui peut être générer soit avec un objet *Counter* pour créer un déroulement séquentielle, soit avec un objet *RandInt* pour créer un déroulement aléatoire. Un deuxième argument (*fname*) spécifie le nom générique des fonctions qui seront appelées. Lorsque *Score* reçoit un nouvel entier, la fonction portant le nom donné en *fname*, avec comme suffixe l'entier en question, est appelée. Voici un script qui illustre la structure d'une organisation séquentielle :

```
1   from pyo import *
2   s = Server().boot()
3
4   def event_0():
5       print("Appel de la fonction 0")
6
7   def event_1():
8       print("Appel de la fonction 1")
9
10  def event_2():
11      print("Appel de la fonction 2")
12
13  def event_3():
14      print("Appel de la fonction 3")
15
16  met = Metro(time=1).play()
17  count = Counter(met, min=0, max=4)
18  score = Score(count, fname="event_")
19
20  s.gui(locals())
```

scripts/chapitre_06/05_score.py

**Variante, avec l'objet *Score*, de l'exercice précédemment créé avec l'objet *Pattern***

Reprenons nos 3 sources de l'exercice précédent et, cette fois-ci, organisons les événements dans le temps. Nous allons tout d'abord créer une fonction indépendante pour le contrôle de chacune des sources. Nous en profiterons pour glisser un petit élément de hasard à chaque fois qu'une source doit démarrer. Si la fonction est appelée avec la valeur 1 en argument (valeur par défaut), le son joue, sinon, il arrête :

```
def play_snd(state=1):
    if state == 1:
        snd.speed = [random.choice([.5,.67,.75,1,1.25,1.5]) for i in range(2)]
        snd_amp.play()
    else:
        snd_amp.stop()

def play_fm(state=1):
    if state == 1:
        freq = random.randint(0,7)*25+50
        fm.carrier = [freq*random.uniform(.99,1.01) for i in range(5)]
        fm_amp.play()
    else:
        fm_amp.stop()

def play_adsyn(state=1):
    if state == 1:
        freq = random.randint(0,7)*25+50
        adsyn.freq = [freq*random.uniform(.99,1.01) for i in range(20)]
        adsyn_amp.play()
    else:
        adsyn_amp.stop()
```

On définit ensuite les fonctions qui seront appelées à tour de rôle, ce qui constituera notre séquence d'événements :

```
def event_0():
    play_fm()
def event_1():
    pass
def event_2():
    play_snd()
def event_3():
    play_adsyn()
    play_fm(0)
def event_4():
    pass
def event_5():
    play_snd(0)
    play_fm()
def event_6():
    play_adsyn(0)
def event_7():
    play_snd()
def event_8():
    play_fm(0)
    play_adsyn()
def event_9():
    pass
def event_10():
    play_snd(0)
    play_adsyn(0)
```

```
    met.stop()
```

Finalement, un métronome envoie des clicks à un objet *Counter*, qui avance de 1 à chaque nouveau click, et le compte est donné à un objet *Score* qui appelle les différentes fonctions à tour de rôle.

```
met = Metro(time=5).play()
count = Counter(met, min=0, max=11)
score = Score(count, fname="event_")
```

Au prochain cours, nous élaborerons plus en détail sur le concept de "trigger", le signal audio qui est au coeur de la synchronicité des événements dans *pyo*. Ce signal est généré, entre autre, par l'objet *Metro*.

```
from pyo import *
import random

s = Server().boot()

snd_amp = Fader(fadein=5, fadeout=5, dur=0)
snd = SfPlayer(SNDS_PATH + "/accord.aif", speed=[.998,1.003],
               loop=True, mul=snd_amp*.35).out()
fm_amp = Fader(fadein=5, fadeout=5, dur=0)
fm = FM(carrier=[99,99.7,100,100.4,100.9], ratio=.4987,
                index=8, mul=fm_amp*.1).out()
adsyn_amp = Fader(fadein=5, fadeout=5, dur=0)
adsyn = SineLoop(freq=[random.uniform(145,155) for i in range(20)],
                 feedback=.15, mul=adsyn_amp*.05).out()

def play_snd(state=1):
    if state == 1:
        snd.speed = [random.choice([.5,.67,.75,1,1.25,1.5]) for i in range(2)]
        snd_amp.play()
    else:
        snd_amp.stop()

def play_fm(state=1):
    if state == 1:
        freq = random.randint(0,7)*25+50
        fm.carrier = [freq*random.uniform(.99,1.01) for i in range(5)]
        fm_amp.play()
    else:
        fm_amp.stop()

def play_adsyn(state=1):
    if state == 1:
        freq = random.randint(0,7)*25+50
        adsyn.freq = [freq*random.uniform(.99,1.01) for i in range(20)]
        adsyn_amp.play()
    else:
        adsyn_amp.stop()

def event_0():
```

```
    play_fm()
def event_1():
    pass
def event_2():
    play_snd()
def event_3():
    play_adsyn()
    play_fm(0)
def event_4():
    pass
def event_5():
    play_snd(0)
    play_fm()
def event_6():
    play_adsyn(0)
def event_7():
    play_snd()
def event_8():
    play_fm(0)
    play_adsyn()
def event_9():
    pass
def event_10():
    play_snd(0)
    play_adsyn(0)
    met.stop()

met = Metro(time=5).play()
count = Counter(met, min=0, max=11)
score = Score(count, fname="event_")

s.gui(locals())
```

scripts/chapitre_06/06_score_audio.py

**Bonus** : petit exemple de l'utilisation de l'objet *Pattern* contrôlant une banque de filtres.

```
from pyo import *
import random

s = Server().boot()

amp = Fader(fadein=0.25, fadeout=0.25, dur=0, mul=0.3).play()
noz = Noise(mul=amp)
fbank = ButBP(noz, freq=[250,700,1800,3000], q=40, mul=4).out()

def change():
    f1 = random.uniform(200,500)
    f2 = random.uniform(500,1000)
    f3 = random.uniform(1000,2000)
    f4 = random.uniform(2000,4000)
    fbank.freq = [f1, f2, f3, f4]

lfo = Sine(.1, mul=.5, add=.75)
pat = Pattern(function=change, time=lfo).play()
```

```
19
20 s.gui(locals())
```

scripts/chapitre_06/07_pattern_filtres.py

# Chapitre 7

# Gestion des répertoires et révision

## 7.1 Lire et écrire des fichiers sous Python

La fonction *open* retourne un objet avec lequel on peut lire ou écrire un fichier sur le disque. Deux arguments sont nécessaires, le premier est un *string* indiquant le nom du fichier à lire ou à écrire, tandis que le deuxième indique le mode, c'est à dire la façon dont on veut interagir avec le fichier. Si le fichier à lire se trouve dans le répertoire courant, il peut être appelé simplement par son nom, sinon, il faut donner le lien au complet. Le répertoire courant est celui dans lequel a été lancé la commande 'python' et peut être retrouvé avec la méthode *getcwd* du module *os* (os.getcwd()). Les modes possibles, identifiés par un *string*, sont :

— **'r'** : Ouvre un fichier en lecture seulement.

— **'w'** : Ouvre un fichier en écriture seulement (Si un fichier du même nom existe déjà, il sera effacé).

— **'a'** : Ouvre un fichier en mode d'ajout. Les nouvelles données seront ajoutées à la suite des données déjà présentes.

— **'r+'** : Ouvre un fichier en lecture et en écriture.

Sur les systèmes Windows et Macintosh, si un 'b' est ajouté au mode, le fichier sera ouvert en format binaire. Ce qui donne les modes 'rb', 'wb' et 'r+b'.

### 7.1.1 Méthodes d'un objet fichier

La méthode ***read*** retourne un *string* contenant tout le texte contenu dans le fichier. Si la lecture est déjà rendue à la fin du fichier, la méthode ***read*** retourne un *string vide*.

```
>>> f = open('mon_fichier.txt', 'r')
>>> f.read()
'Ceci est la premiere ligne du fichier.\nEt ceci est la seconde!\n'
>>> f.read()
''
>>>
```

** À noter le symbole du retour de chariot. **

```
\n
```

La méthode ***readline*** lit et retourne une ligne du fichier à la fois. Comme pour la méthode ***read***, si la lecture est rendue à la fin du fichier, un *string* vide sera retourné.

```
>>> f = open('mon_fichier.txt', 'r')
>>> f.readline()
'Ceci est la premiere ligne du fichier.\n'
>>> f.readline()
'Et ceci est la seconde!\n'
>>> f.readline()
''
>>>
```

Il est possible de lire toutes les lignes d'un fichier et de les mémoriser sous forme de liste avec la méthode ***readlines***.

```
>>> f = open('mon_fichier.txt', 'r')
>>> f.readlines()
['Ceci est la premiere ligne du fichier.\n', 'Et ceci est la seconde!\n']
>>>
```

La méthode ***write*** permet d'écrire des lignes dans un fichier.

```
>>> f = open('mon_fichier.txt', 'w')
>>> f.write('Hello world!\n')
>>>
```

Lorsque les opérations sur un fichier sont terminées, il faut refermer le fichier avec la méthode ***close***.

```
>>> f = open('mon_fichier.txt', 'w')
>>> f.write('Hello world!\n')
>>> f.close()
>>> f = open('mon_fichier.txt', 'r')
>>> str = f.read()
>>> str
'Hello world!\n'
>>> f.close()
```

### 7.1.2 Exemple concret d'écriture et de lecture d'un fichier texte.

L'exemple suivant illustre la sauvegarde d'une séquences de notes, générée à l'aide d'un algorithme, dans un fichier texte et la relecture de celui-ci à des fins musicales. L'avantage de sauvegarder la séquence dans un fichier texte est que si la séquence générée est intéressante, elle est toujours accessible en format texte, même après avoir quitté le programme.

```
################################################################################
# --- Ecriture et lecture de fichier sur le disque dur. ---                    #
# 1 - Generation de 4 melodies de 16 notes chacune dans un fichier.            #
# 2 - Relecture du fichier et conversion des lignes en liste d'entiers.        #
# 3 - Petit algo qui boucle sur les listes de notes et controle un synth.      #
################################################################################
from pyo import *
import random

s = Server().boot()

# Do mineur harmonique (on pige les notes dans cette liste a l'aide d'un index).
scale = [36,38,39,41,43,44,47,48,50,51,53,55,56,59,60,62,63,65,67,68,71,72]
# Reference a la longueur de la liste-1 parce que randint inclut le maximum.
maxn = len(scale) - 1
# Index de depart pour la generation des melodies.
index = random.randint(0, maxn)

# Fonction de marche aleatoire (retourne +/- 3 par rapport a l'index precedent).
def drunk():
    # On veut modifier la variable global "index", et non en creer une locale.
    global index
    # On additionne a "index" un entier pige au hasard entre -3 et 3.
    index += random.randint(-3,3)
    # Assure que index reste toujours entre 0 et maxn (longueur de la liste).
    if index < 0:
        index = abs(index)
    elif index > maxn:
        index = maxn - (index - maxn)

### Ecriture d'un fichier texte ###

# Ouvre un fichier en mode ecriture.
f = open("notes.txt", "w")
# Genere 4 lignes de 16 valeurs.
for i in range(4):
    for j in range(16):
        # Change la variable globale "index".
        drunk()
        # Ecrit le nouvel index, suivi d'un espace, dans le fichier.
        f.write(str(index) + " ")
    # Apres chaque groupe de 16 valeurs, on change de ligne.
    f.write("\n")

# Ferme le fichier
f.close()

```

```
### Lecture d'un fichier texte ###

# Ouvre le fichier en mode lecture et recupere les melodies.
f = open("notes.txt", "r")
# Chaque ligne est un element (un string contenant 16 chiffres) d'une liste.
mel_list = f.readlines()
# On ferme le fichier.
f.close()

### Conversion d'une chaine de caracteres en liste d'entiers ###

# On enleve les retours de chariot.
mel_list = [l.replace("\n", "") for l in mel_list]
# On separe les donnees dans chaque liste.
mel_list = [l.split() for l in mel_list]
# On convertit chacun des strings en entier.
for i in range(4):
    for j in range(16):
        mel_list[i][j] = int(mel_list[i][j])

# Ici, mel_list contient 4 listes de 16 index pour lire dans la liste "scale".

############## Processus audio (synth old school) ################

# Enveloppe d'amplitude
f = Fader(fadein=.005, fadeout=.05, dur=.125, mul=.1)
# 10 frequences bidons (pour initialiser un chorus)
freqs = [100]*10
# 10 ondes carrees.
a = LFO(freq=freqs, sharp=.75, type=2, mul=f)
# synth -> disto -> reverb
pha = Disto(a.mix(2), drive=.85, slope=.95, mul=.2)
rev = WGVerb(pha, feedback=.75, cutoff=3000, bal=.2).out()

############## Section controles ################

# Variables globales...
mel = 0 # Laquelle des 4 melodies.
count = 0 # Compte des temps dans la melodie.
def new_note():
    global mel, count
    # Recupere l'index courant (dans la melodie "mel", au temps "count".
    ind = mel_list[mel][count]
    # Recupere la note midi dans la liste "scale".
    midi = scale[ind]
    # Calcul de la frequence en Hertz.
    freq = midiToHz(midi)
    # Chorus de 10 valeurs autour de la frequence pigee.
    freqs = [freq*random.uniform(.99,1.01) for i in range(10)]
    # Amplitude plus grande aux 4 temps (accent sur les temps forts).
    if (count % 4) == 0:
        f.mul = .1
    else:
        f.mul = .03
```

```
    # Incremente le compte (pour avancer au prochain appel de la fonction).
    count += 1
    # Actions a la fin de la mesure:
    if count == 16:
        # Remettre le compte a 0.
        count = 0
        # 50% du temps on pige une nouvelle melodie.
        if random.randint(0,1) == 0:
            mel = random.randint(0,3)
            print("nouvelle liste d'index :", mel)
    # Assigne les frequences aux oscillateurs et demarre l'enveloppe.
    a.freq = freqs
    f.play()

# Appelle "new_note" 8 fois par seconde.
pat = Pattern(time=.125, function=new_note).play()

s.gui(locals())
```

scripts/chapitre_07/01_fichier_texte.py

## 7.2 Révision

Générateur de grains avec délais récursifs. Ce script utilise tous les aspects développés jusqu'ici. Étudiez chaque partie attentivement !

```
from pyo import *
import random

s = Server().boot()

################################################################################
### SNDS_PATH est un string provenant du module pyo qui pointe sur le ###
### dossier ou sont installes les sons fournis avec pyo. N'utilisez   ###
### jamais cette constante pour lire vos propre sons.                 ###
################################################################################
son = SNDS_PATH + "/transparent.aif"

# Garde en memoire la duree du son (2e element de la liste que retourne sndinfo).
son_dur = sndinfo(son)[1]
print("Duree du son :", son_dur)

# Liste de notes midi pour restreindre les transposition permises.
mnotes = [48, 50, 53, 55, 57, 60, 62, 65, 67, 69, 72]
# Conversion en facteur de transposition autour de la note 60.
transpos = [midiToTranspo(x) for x in mnotes]

####### Processus audio #######

# Enveloppe et mise en place d'un lecteur de son.
f = Fader(fadein=.005, fadeout=.01, dur=.02)
sf = SfPlayer(son, speed=1, mul=f)

```

```
28 # Banque de delais (4 delais avec feedback differents).
29 dls = Delay(sf, delay=[.1,.2,.3,.4], feedback=[.25,.5,.35,.75],
30             maxdelay=1, mul=.25)
31 # Disposition dans l'espace stereo de chacun des delais.
32 pan = Pan(dls, pan=[0,.33,1,.66], spread=.25)
33 # Reverbe generale, mix(2) pour une reverbe stereo.
34 rev = WGVerb(pan.mix(2), feedback=.8, cutoff=4000, bal=.15).out()
35
36 ####### Section controle #######
37
38 # Gestion des methodes de controle. Voir la fonction "new_stream" pour
39 # la signification de ces variables. Valeurs possibles, 0 ou 1.
40 offset_meth = 0
41 speed_meth = 0
42
43 # Position du grain dans le son en secondes (pour la methode 1 a offset_meth).
44 start = 0
45
46 # Increment sur la position en seconde. Appliquee a chaque appel de new_stream.
47 start_inc = 0.01
48
49 # Fonction appelee periodiquement par l'objet Pattern (plus bas).
50 def new_stream():
51     # variable globale puisque qu'on veut la modifier.
52     global start
53
54     # si offset_meth vaut 0, on pige un point de lecture au hasard.
55     # Sinon, on avance dans le son en fonction de l'increment. On ne
56     # declare pas l'increment global puisqu'on ne le modifie pas!
57     if offset_meth == 0:
58         sf.offset = random.uniform(0, son_dur-.1)
59     else:
60         start += start_inc
61         if start >= (son_dur-.1): # Rendu a la fin du son, on revient au debut.
62             start = 0
63         sf.offset = start
64
65     # si speed_meth vaut 0, on pige une transposition dans la liste de transpos.
66     # Sinon, on pige une legere deviation autour de 1.
67     if speed_meth == 0:
68         sf.speed = random.choice(transpos)
69     else:
70         sf.speed = random.uniform(.99, 1.01)
71
72     # Nouvelle duree de grain.
73     f.dur = random.uniform(.02, .1)
74
75     # Affichage des nouvelles valeurs.
76     print("offset: %.2f, speed: %.2f, dur: %.2f" % (sf.offset, sf.speed, f.dur))
77
78     # On lance la lecture du son et de l'enveloppe d'amplitude.
79     sf.play()
80     f.play()
81
```

```
82 # Variations sur la frequence des appels de la fonction new_stream.
83 vtime = Choice(choice=[.2, .4, .8, 1.2, 2.4], freq=1./2.4)
84
85 # Pattern appelle de facon periodique la fonction donnee en argument
86 # (new_stream *sans parentheses*). On *doit* appeler la methode play()
87 # de l'objet Pattern pour l'activer.
88 pat = Pattern(time=vtime, function=new_stream).play()
89
90 s.gui(locals())
```

scripts/chapitre_07/02_grains_delay.py

# Chapitre 8

# Les Dictionnaires

Deux nouveaux types de données sont présentés dans ce chapitre : le **tuple** et le **dictionnaire**.

## 8.1 Le *Tuple*

Un *tuple*, tout comme la liste, est une suite de valeurs séparées par des virgules. Par contre, un *tuple* est délimité par des parenthèses et est immuable. Particulièrement utile pour créer des listes qui ne doivent en aucun cas être modifiées, le *tuple* est aussi plus rapide d'accès que la liste.

```
>>> tup = (21, 22, 23, 24, 25, 26)
>>> tup[0]
21
>>> tup[2:5]
(23,24,25)
>>> tup2 = (21, 1.33, 'bonjour', [1,2,3])
>>> tup2[1]
1.330000000000001
>>> tup2[2:4]
('bonjour', [1,2,3])
>>> tup2[1] = 'Allo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

## 8.2 Le *dictionnaire*

Le dictionnaire est une table de données, définie entre accolades {}, sur base de pairs clé - valeur. Les pairs sont séparées par des virgules et la clé est séparée de sa valeur par le symbole « deux-points » ( : ).

```
mon_dict = {1: 100, 2: 101, 3: 102}
```

Tous les objets valides en Python peuvent servir de clé dans un dictionnaire (entier, décimale, *string*, liste, tuple, objet, function, etc.) On accède aux valeurs dans un dictionnaire en l'appelant et en lui donnant la clé entre crochets.

```
>>> mon_dict = {'un'  : 123,
                12    : [1, 2, 3, 4, 5, 6],
                (1, 2): 'Hello World!'}

>>> print(mon_dict['un'])
123
>>> print(mon_dict[12])
[1, 2, 3, 4, 5, 6]
>>> print(mon_dict[(1,2)])
Hello world!
```

Pour des clés qui sont de simples *strings*, il est possible de spécifier les pairs clé-valeur en donnant des mots-clés à la méthode « constructeur » (cet aspect sera éclaircie au prochain cours lorsque nous aborderons le concept de classe) de la classe dictionnaire.

```
>>> mon_dict = dict(Montreal=514, Quebec=418, Sherbrooke=819)
>>> print(mon_dict)
{'Montreal': 514, 'Quebec': 418, 'Sherbrooke': 819}
```

Un dictionnaire n'est pas une séquence, les éléments ne sont donc pas classés en ordre, ce qui signifie qu'il est impossible de prédire l'ordre d'arrivé des éléments, dans une boucle **for** par exemple.

```
>>> for cle in mon_dict.keys():
...     print(cle, mon_dict[cle])
...
Sherbrooke 819
Quebec 418
Montreal 514
```

La méthode *keys*, qui retourne une liste contenant toutes les clés du dictionnaire, peut facilement être utilisée pour faire le tour de tous les éléments du dictionnaire. Si l'ordre est important, il suffit de remettre la liste en ordre croissant avec la fonction *sorted*.

```
>>> for cle in sorted(mon_dict.keys()):
...     print(cle, mon_dict[cle])
...
Montreal 514
Quebec 418
Sherbrooke 819
```

### 8.2.1 Opérations sur les dictionnaires

Quelques opérations sur un dictionnaire :

— Ajout d'éléments :

```
>>> dict = {'Montreal': 514, 'Quebec': 418}
>>> dict['Sherbrooke'] = 819
>>> print(dict['Sherbrooke'])
819
```

— Suppression d'éléments :

```
>>> del dict['Quebec']
>>> dict
{'Montreal': 514, 'Sherbrooke': 819}
```

— Lister des clés ou des valeurs :

```
>>> dict.keys()
['Montreal', 'Sherbrooke']
>>> dict.values()
[514, 819]
```

— Lister tous les éléments :

```
>>> dict.items()
[('Montreal', 514), ('Sherbrooke', 819)]
```

— Tester la présence d'une clé (avec le mot-clé **in**) :

```
>>> 'Sherbrooke' in dict
True
```

— Copier un dictionnaire (créer une copie indépendante) :

```
>>> new_dict = dict.copy()
>>> new_dict['Quebec'] = 418
>>> print(new_dict)
{'Montreal': 514, 'Quebec': 418, 'Sherbrooke': 819}
>>> print(dict)
{'Montreal': 514, 'Sherbrooke': 819}
```

À noter que la méthode *copy()* fonctionne seulement avec des dictionnaires relativement simples, c'est-à-dire des dictionnaires qui ne contiennent pas de « conteneurs » parmi leurs items. Si, par exemple, une liste est associée à une clé quelconque dans le dictionnaire, comme la copie n'est pas récursive, la liste ne sera pas indépendante de la liste du dictionnaire à la source. Ceci implique qu'une modification appliquée à la liste se reflétera dans les deux dictionnaires, bien que ceux-ci soient indépendants. La liste demeure un seul objet en mémoire avec deux références pointant sur cet espace-mémoire.

```
>>> d1 = {'p1': [0,2,5,7,10]}
>>> d2 = d1.copy()
>>> print("d1 =", d1)
d1 = {'p1': [0, 2, 5, 7, 10]}
>>> d2['p1'][4] = 9
>>> print(mon_dict"d2 =", d2)
d2 = {'p1': [0, 2, 5, 7, 9]}
>>> print("d1 =", d1)
d1 = {'p1': [0, 2, 5, 7, 9]}
```

La solution à ce problème consiste à faire appel au module *copy*, dans lequel on trouvera la fonction *deepcopy* permettant de copier de façon récursive des structures complexes. Avec cette fonction, non seulement le conteneur principal est copié, mais aussi tous les conteneurs qui se trouvent parmi les items du dictionnaire.

```
>>> import copy
>>> d1 = {'p1': [0,2,5,7,10]}
>>> d2 = copy.deepcopy(d1)
>>> print ("d1 =", d1)
d1 = {'p1': [0, 2, 5, 7, 10]}
>>> d2['p1'][4] = 9
>>> print("d2 =", d2)
d2 = {'p1': [0, 2, 5, 7, 9]}
>>> print("d1 =", d1)
d1 = {'p1': [0, 2, 5, 7, 10]}
```

Tutoriel sur les différentes structures de data.

### 8.2.2 Construction d'un histogramme à l'aide d'un dictionnaire

En supposant la phrase suivante : *la programmation en python est vraiment amusante.*

La construction d'un histogramme, représentant la fréquence de répétition de chacune des lettres, est très simple à faire si on se base sur un dictionnaire.

```
>>> text = "la programmation en python est vraiment amusante"
>>> lettres = {}
>>> for c in text:
...     lettres[c] = lettres.get(c, 0) + 1
...
>>> print(lettres)
```

```
{'a': 6, ' ': 6, 'e': 4, 'g': 1, 'i': 2, 'h': 1,
'm': 4, 'l': 1, 'o': 3, 'n': 5, 'p': 2, 's': 2,
'r': 3, 'u': 1, 't': 5, 'v': 1, 'y': 1}
```

Dans un premier temps, nous créons un dictionnaire vide : *lettres*. Ensuite, nous parcourons la chaîne de caractères, élément par élément. Pour chaque caractère, nous appelons la méthode *get* de la classe dictionnaire. Cette méthode retourne la valeur associée à la clé indiquée au premier paramètre. Si cette clé n'est pas présente, la méthode renvoie une valeur par défaut définie au deuxième paramètre, dans ce cas-ci, la valeur 0. Ainsi, pour chaque caractère, nous questionnons le dictionnaire afin de savoir combien de fois a passé ce caractère depuis le début de la boucle, nous ajoutons 1 à la réponse et remplaçons par cette valeur l'ancienne valeur associée à ce caractère.

Si on veut obtenir le résultat pour chaque lettre dans l'ordre alphabétique, il suffit de convertir notre dictionnaire en une liste de *tuples*, avec la méthode *items*, puis de passer la liste dans la fonction *sorted*, pour les placer dans l'ordre :

```
>>> lettres_triees = sorted(lettres.items())
>>> print(lettres_triees)
[(' ', 6), ('a', 6), ('e', 4), ('g', 1), ('h', 1), ('i', 2),
('l', 1), ('m', 4), ('n', 5), ('o', 3), ('p', 2), ('r', 3),
('s', 2), ('t', 5), ('u', 1), ('v', 1), ('y', 1)]
```

## 8.3 Gestion des événements dans le temps par l'envoi de *triggers*

### 8.3.1 Qu'est-ce qu'un *trigger* ?

Un *trigger*, ou *trig*, est une impulsion, c'est-à-dire un signal audio dans lequel un échantillon ayant une valeur de 1.0 est entouré d'échantillons de valeurs 0.0. La raison d'être de ce type de signal est de pourvoir *pyo* d'un système de gestion des événements à haute résolution temporelle. Un *trigger* étant un signal audio, il possible de créer des traitements parallèles avec une synchronisation à l'échantillon près. Il y a deux types d'objets dans la catégorie *triggers*, les objets qui génèrent des *trigs* et ceux qui y réagissent.

#### Objets générant des *trigs*

— Beat génère une séquence de *trigs* de façon algorithmique.

— Change envoie un *trig* lorsqu'un signal audio en entrée change de valeur.

— Cloud envoie des *trigs*, sans rythmique particulière, en fonction d'un paramètre de densité.

— Metro génère des *trigs* de façon synchrone.

— Select envoie un *trig* lorsque le signal en entrée correspond à un entier donné en argument.

— Thresh envoie un *trig* lorsqu'un signal en entrée croise un seuil donnée en argument.

— Trig envoie un seul *trig* à l'appel de la méthode *play*.

**Objets réagissant aux *trigs***

— Counter incrémente un compte d'entier.
— TrigChoice pige une valeur dans une liste donnée en argument.
— TrigEnv démarre la lecture d'une table d'échantillons.
— TrigExpseg démarre la lecture d'une enveloppe exponentielle.
— TrigFunc appelle une fonction python.
— TrigLinseg démarre la lecture d'une enveloppe linéaire.
— TrigRand génère une valeur pseudo-aléatoire.
— TrigXnoise génère une valeur selon différents algorithmes pseudo-aléatoires.
— TrigXnoiseMidi génère une note Midi selon différents algorithmes pseudo-aléatoires.

Consulter la catégorie triggers du manuel *pyo* pour une liste complète des objets participant à la gestion des *trigs*.

### 8.3.2 Séquence d'événements

Dans un premier temps, nous nous concentrerons sur les objets permettant de créer des séquences d'événements, c'est-à-dire *Metro* et *Beat*. Voici, avec un métronome, comment changer de façon périodique, la fréquence d'un oscillateur :

```
from pyo import *

s = Server().boot()

# Ne pas oublier d'appeler la methode .play() pour les objets Metro et Beat
met = Metro(time=.125).play()
t = TrigRand(met, min=400, max=1000)
a = SineLoop(freq=t, feedback=0.05, mul=.3).out()

s.gui(locals())
```

scripts/chapitre_08/01_metro_rand.py

Même procédé mais cette fois-ci en pigeant les valeurs dans une gamme prédéfinie :

```
from pyo import *

s = Server().boot()

notes = [midiToHz(x) for x in [60,62,64,65,67,69,71,72]]
met = Metro(time=.125).play()
t = TrigChoice(met, choice=notes, port=0.005)
a = SineLoop(freq=t, feedback=0.07, mul=.3).out()

s.gui(locals())
```

scripts/chapitre_08/02_metro_choice.py

L'objet *TrigChoice* agit de la même façon que l'objet *Choice*. Par contre, il attend un *trig* pour piger une valeur au lieu d'effectuer sa pige en fonction d'une fréquence donnée. L'argument *port* permet de spécifier un temps de rampe entre la valeur actuelle et la valeur pigée.

### Utilisation d'échantillons placés en mémoire dans une table

Afin d'obtenir l'effet d'une nouvelle note jouée, il est nécessaire d'appliquer une enveloppe d'amplitude sur le signal. L'objet *TrigEnv* permet de démarrer la lecture du contenu d'une table d'échantillons sur réception d'un *trig* (nous élaborerons plus en détail sur le type *PyoTableObject* prochainement). Nous utiliserons ici quelques objets de la classe *PyoTableObject* afin de mettre en mémoire différentes enveloppes d'amplitude.

— LinTable Construit une enveloppe à partir de segments de droite.
— ExpTable Construit une enveloppe à partir de segments exponentiels.
— HannTable Génère une enveloppe en forme de cloche (Hanning).

*LinTable* et *ExpTable* attendent une liste de *tuples* représentant chacun des points de l'enveloppe.

```
# Enveloppe ADSR (attack - decay - sustain - release)
env = LinTable([(0,0), (100,1), (500,.5), (5000,.5), (8191,0)], size=8192)
```

La première valeur du *tuple* représente l'emplacement, en échantillons, où situer le point et la deuxième valeur est l'amplitude du point. Ainsi, dans l'exemple précédent, pour une table de 8192 échantillons, nous avons les points suivants :

— **point 1** : valeur 0 à l'échantillon 0 (premier emplacement).
— **point 2** : valeur 1 à l'échantillon 100 (fin de l'attaque).
— **point 3** : valeur 0.5 à l'échantillon 500 (fin du *decay*).
— **point 4** : valeur 0.5 à l'échantillon 5000 (fin de la tenue).
— **point 5** : valeur 0 à l'échantillon 8191 (Relâche jusqu'à la fin de la table).

Ajoutons l'enveloppe dans le script précédent.

```
1  from pyo import *
2
3  s = Server().boot()
4
5  env = LinTable([(0,0), (100,1), (500,.5), (5000,.5), (8191,0)], size=8192)
6  notes = [midiToHz(x) for x in [60,62,64,65,67,69,71,72]]
7
8  met = Metro(time=.125).play()
9  t = TrigChoice(met, choice=notes, port=0.005)
10 amp = TrigEnv(met, table=env, dur=.125, mul=.5)
11 a = SineLoop(freq=t, feedback=0.07, mul=amp).out()
12
13 s.gui(locals())
```

scripts/chapitre_08/03_metro_env.py

Vous pouvez aisément tester l'effet d'une enveloppe exponentielle en remplaçant *LinTable* par *ExpTable*. Pour obtenir une enveloppe en forme de cloche, il suffit simplement de créer un objet *HannTable* sans paramètre. Dans le script suivant, un deuxième métronome, beaucoup plus lent, a été ajouté pour déclencher une enveloppe globale sur la génération de notes.

```
from pyo import *

s = Server().boot()

env = LinTable([(0,0), (100,1), (500,.5), (5000,.5), (8191,0)], size=8192)
globenv = HannTable()

notes = [midiToHz(x) for x in [60,62,64,65,67,69,71,72]]

globmet = Metro(time=5).play()
globamp = TrigEnv(globmet, table=globenv, dur=5)

met = Metro(time=.125).play()
t = TrigChoice(met, choice=notes, port=0.005)
amp = TrigEnv(met, table=env, dur=.125, mul=.5)
a = SineLoop(freq=t, feedback=0.07, mul=amp*globamp).out()

s.gui(locals())
```

scripts/chapitre_08/04_metro_env_glob.py

Un des principaux dilemmes en synthèse et traitement des sons est la gestion de la polyphonie, c'est-à-dire la duplication des processus. Un exemple rapide pour démontrer l'importance de la polyphonie consiste simplement à allonger la durée de l'enveloppe du script précédent à .25 seconde afin d'obtenir des notes plus longues, qui se chevauchent. Qu'en est-il du résultat sonore ?

Il y a des clics dans le son puisque l'on redémarre une enveloppe qui n'avait pas encore terminée sa course. Pour allonger les notes et créer un chevauchement, il faut absolument avoir plus d'une instance du même processus. Cela assurera qu'une enveloppe est bien terminée avant d'être relancée, évitant ainsi une discontinuité dans la trajectoire d'amplitude.

Les objets qui génèrent des séquences de *trigs* (Metro, Beat et Cloud) ont tous un argument *poly* permettant de définir combien de *streams* audio seront asservis à une génération d'impulsions.

```
>>> a = Metro(time=.125, poly=4)
>>> print(len(a))
4
>>>
```

Si notre métronome gère 4 *streams* de polyphonie, la suite d'objets qui en découlent géreront forcément au moins 4 *streams* de polyphonie chacun. On peut donc dupliquer un processus sonore au complet simplement en spécifiant un certain nombre de voix de polyphonie à l'objet *Metro*. Pour une durée d'enveloppe de .25 seconde, nous avons besoin de seulement 2 voix de polyphonie.

```
from pyo import *

s = Server().boot()

env = LinTable([(0,0), (100,1), (500,.5), (5000,.5), (8191,0)], size=8192)
globenv = HannTable()

notes = [midiToHz(x) for x in [60,62,64,65,67,69,71,72]]

globmet = Metro(time=5).play()
globamp = TrigEnv(globmet, table=globenv, dur=5)

met = Metro(time=.125, poly=2).play()
t = TrigChoice(met, choice=notes, port=0.005)
amp = TrigEnv(met, table=env, dur=.25, mul=.5)
a = SineLoop(freq=t, feedback=0.07, mul=amp*globamp).out()

s.gui(locals())
```

scripts/chapitre_08/05_metro_stereo.py

### Lecture d'un fichier sonore placé en mémoire dans une table

Avec l'objet *TrigEnv*, il est très simple de faire jouer un son en boucle à une fréquence déterminée par la vitesse d'un métronome. Nous allons cette fois-ci placer un son dans une table avec l'objet SndTable et activer la lecture avec une séquence de *trigs*. La méthode *getDur* de l'objet *SndTable* permet de récupérer la longueur de la table en secondes.

```
from pyo import *

s = Server().boot()

snd = SndTable(SNDS_PATH + "/accord.aif")
dur = snd.getDur()
print("Duree:", dur)

met = Metro(time=dur).play()
out = TrigEnv(met, table=snd, dur=dur).out()

s.gui(locals())
```

scripts/chapitre_08/06_sndtable.py

On peut changer la hauteur du son en modifiant la durée de lecture de la table :

```
snd = SndTable(SNDS_PATH + "/accord.aif")
dur = snd.getDur()
print("Duree:", dur)

met = Metro(time=dur*2).play()
out = TrigEnv(met, table=snd, dur=dur*2).out()
```

Si la durée de lecture de la table dépasse la vitesse du métronome, on ajoute des voix de polyphonie afin d'éviter les clics !

```
snd = SndTable(SNDS_PATH + "/accord.aif")
dur = snd.getDur()
print("Duree:", dur)

met = Metro(time=dur*2, poly=2).play()
out = TrigEnv(met, table=snd, dur=dur*2.5).out()
```

## 8.4 Création d'algorithmes musicaux

### Rythme

Dans cette section, nous allons explorer certains objets d'algorithmie disponibles dans la librairie *pyo*. Reprenons un script similaire à l'exemple de la page précédente, c'est-à-dire un métronome qui active une enveloppe d'amplitude et pige une nouvelle fréquence pour un oscillateur.

```
freqs = [midiToHz(x) for x in [60,62,64,65,67,69,71,72]]
env = LinTable([(0,0), (300,1), (1000,.5), (3000,.5), (8191,0)], size=8192)

met = Metro(time=.125).play()

amp = TrigEnv(met, table=env, dur=.125, mul=.5)
fr = TrigChoice(met, choice=freqs)
osc = SineLoop(freq=fr, feedback=.08, mul=amp).out()
```

Nous allons dans un premier temps utiliser l'objet *Beat* qui est conçu pour générer des séquences rythmiques de façon algorithmique en fonction de poids donnés en argument. Les paramètres *w1*, *w2* et *w3* contrôlent respectivement le pourcentage de chance d'être présent des temps forts, intermédiaires et faibles. Le paramètre *taps* indique le nombre de temps dans la mesure (1 temps = une durée donnée au paramètre *time*). Par exemple, pour obtenir une mesure en 4/4 ne contenant que les temps forts (les noires), on écrirait ceci :

```
met = Beat(time=.125, taps=16, w1=100, w2=0, w3=0)
```

Pour rajouter 50% des temps intermédiaires (les croches) :

```
met = Beat(time=.125, taps=16, w1=100, w2=50, w3=0)
```

Et Finalement, 30% des temps faibles (toutes les double-croches restantes) :

```
met = Beat(time=.125, taps=16, w1=100, w2=50, w3=30)
```

Remplaçons le métronome du script par un objet *Beat* :

```
freqs = [midiToHz(x) for x in [60,62,64,65,67,69,71,72]]
env = LinTable([(0,0), (300,1), (1000,.5), (3000,.5), (8191,0)], size=8192)

met = Beat(time=.125, taps=16, w1=90, w2=50, w3=30).play()

amp = TrigEnv(met, table=env, dur=.125, mul=.5)
fr = TrigChoice(met, choice=freqs)
osc = SineLoop(freq=fr, feedback=.08, mul=amp).out()
```

Ce script générera une nouvelle séquence rythmique à chaque exécution. Une nouvelle particularité est à noter ici. L'objet *Beat* ne donne pas que des *trigs* comme signaux, il retourne aussi d'autres informations utiles sous la forme de *streams* audio. On accède aux différents *streams* en utilisant une syntaxe similaire à celle des dictionnaires, en appelant l'info désirée, en format *string*, entre crochets. Voici les différents appels possibles :

— **obj['dur']** : Signal audio contenant la durée de la « note » courante, c'est-à-dire le temps avant le prochain *trig*.

— **obj['amp']** : Signal audio contenant l'amplitude de la « note » courante, en fonction des temps forts, intermédiaires et faibles.

— **obj['end']** : Signal audio contenant un seul *trig* sur le dernier temps de la mesure. Il peut-être utilisé pour s'assurer que les changements de rythme arrivent bien au début d'une mesure.

Nous allons insérer dans notre script les signaux de durée et d'amplitude afin que notre algorithme se comporte de façon un peu plus cohérente :

```
freqs = [midiToHz(x) for x in [60,62,64,65,67,69,71,72]]
env = LinTable([(0,0), (300,1), (1000,.5), (3000,.5), (8191,0)], size=8192)

met = Beat(time=.125, taps=16, w1=90, w2=50, w3=30).play()

amp = TrigEnv(met, table=env, dur=met['dur'], mul=met['amp'])
fr = TrigChoice(met, choice=freqs)
osc = SineLoop(freq=fr, feedback=.08, mul=amp*0.5).out()
```

### Mélodie

Maintenant que nous avons géré les accents et les durées, laissons de côté l'aspect rythmique pour améliorer l'aspect mélodique de notre algorithme. Une mélodie est rarement constituée de notes pigées au hasard, même si la pige s'effectue à l'intérieur d'une gamme. En règle générale, une mélodie est constituée de mouvements (souvent conjoints) vers le haut et vers le bas qui peuvent s'étaler sur plusieurs notes. On y retrouvera aussi des répétitions de certains motifs. C'est donc dire que notre objet *TrigChoice* ne rempli pas vraiment son rôle mélodique, il est trop hasardeux ! Il y a deux objets dans la librairie qui sont plus adéquats pour

générer des cellules mélodiques, *TrigXnoise* et *TrigXnoiseMidi*. Comme nous désirons générer des notes sur des gammes tempérées, nous utiliserons *TrigXnoiseMidi*, qui génère d'abord sa mélodie en notes Midi, puis peut, sur demande, convertir automatiquement en Hertz. Cet objet implémente différents types de distributions aléatoires contrôlées pouvant servir à générer des motifs mélodiques de toutes sortes. Pour en savoir plus sur les distributions aléatoires contrôlées (qui est un peu hors du contexte de ce cours), voici un excellent site avec des schémas représentant les courbes de distributions des algorithmes les plus populaires :

*Gallery of distributions*

L'algorithme qui nous intéresse dans le présent contexte est le numéro 12, un générateur de segments de mélodie en boucle. Remplaçons la ligne du *TrigChoice* par la ligne suivante :

```
fr = TrigXnoiseMidi(met, dist=12, x1=1, x2=.3, scale=1, mrange=(60,84))
```

L'argument *dist* indique le numéro de distribution à utiliser, consultez la liste des algorithmes disponibles dans le manuel. Les arguments *x1* et *x2* changent de rôle en fonction de l'algorithme choisi. Dans ce cas-ci, ils représentent la valeur maximale possible (en fonction de l'ambitus Midi) et la plus grande distance possible entre 2 hauteurs de note. L'argument *scale* indique le format désiré pour les valeurs obtenues (0 = Midi, 1 = Hertz) et l'argument *mrange* est un *tuple* indiquant le registre Midi dans lequel doit se restreindre la pige.

Notre script donne maintenant un profil mélodique plus cohérent, par contre, nous avons perdu notre gamme. Toutes les notes entre 60 et 84 sont maintenant légales. Une technique rudimentaire mais efficace pour pallier à ce problème consiste à choisir la note de la gamme la plus proche de la note pigée. L'objet *Snap* effectue cette tâche :

```
fr = TrigXnoiseMidi(met, dist=12, x1=1, x2=.3, scale=0, mrange=(60,84))
frsnap = Snap(fr, choice=[0,2,3,5,7,8,11], scale=1)
osc = SineLoop(freq=frsnap, feedback=.08, mul=amp*.5).out()
```

L'objet *Snap* fonctionne avec des notes Midi, il ne faut donc pas oublier de remettre l'argument *scale* à 0, pour envoyer des notes Midi. C'est maintenant l'objet *Snap* qui se chargera de la conversion en Hertz, en passant la valeur 1 à son propre argument *scale*. Au paramètre *choice*, on donnera une liste représentant le premier octave de la gamme désirée.

Voici le script en entier :

```
1  from pyo import *
2
3  s = Server().boot()
4
5  env = LinTable([(0,0), (300,1), (1000,.5), (3000,.5), (8191,0)], size=8192)
6
7  met = Beat(time=.125, taps=16, w1=90, w2=50, w3=30).play()
8
9  amp = TrigEnv(met, table=env, dur=met['dur'], mul=met['amp'])
10
11 fr = TrigXnoiseMidi(met, dist=12, x1=1, x2=.3, scale=0, mrange=(48,85))
12 frsnap = Snap(fr, choice=[0,2,3,5,7,8,11], scale=1)
```

```
13
14 osc = SineLoop(freq=frsnap, feedback=.08, mul=amp*.5).out()
15
16 s.gui(locals())
```

scripts/chapitre_08/07_simple_algo.py

### Harmonie

Pour créer un contrepoint à deux voix, il suffit de dupliquer le générateur en place et de modifier les paramètres afin de jouer sur des registres et des rythmiques intéressantes...

```
1  from pyo import *
2
3  s = Server().boot()
4
5  env = LinTable([(0,0), (300,1), (1000,.5), (3000,.5), (8191,0)], size=8192)
6
7  met = Beat(time=.25, taps=16, w1=50, w2=70, w3=50).play()
8  amp = TrigEnv(met, table=env, dur=met['dur'], mul=met['amp'])
9  fr = TrigXnoiseMidi(met, dist=12, x1=1, x2=.3, scale=0, mrange=(48,73))
10 frsnap = Snap(fr, choice=[0,2,3,5,7,8,11], scale=1)
11 osc = SineLoop(freq=frsnap, feedback=.08, mul=amp*.5).out()
12
13 met2 = Beat(time=.25, taps=16, w1=100, w2=40, w3=0).play()
14 amp2 = TrigEnv(met2, table=env, dur=met2['dur'], mul=met2['amp'])
15 fr2 = TrigXnoiseMidi(met2, dist=12, x1=1, x2=.3, scale=0, mrange=(36,61))
16 frsnap2 = Snap(fr2, choice=[0,2,3,5,7,8,11], scale=1)
17 osc2 = SineLoop(freq=frsnap2, feedback=.08, mul=amp2*.5).out(1)
18
19 s.gui(locals())
```

scripts/chapitre_08/08_stereo_algo.py

## 8.5 Exemple : Petit contrepoint algorithmique à 2 voix

```
#!/usr/bin/env python
# encoding: utf-8
from pyo import *
import random

s = Server().boot()

TIME = .2 # Duree d'un tap
# Dictionnaire de modes pour le choix des notes
SCLS = { 'p1': [0,2,5,7,9], 'p2': [0,3,5,7,10], 'p3': [0,2,7,8,11] }

# Envelope d'amplitude
env = LinTable([(0,0), (300,1), (1000,.7), (6000,.7), (8191,0)], size=8192)

# 8 randis a multiplier aux notes pour creer un chorus
cho = Randi(min=.99, max=1.01, freq=[random.uniform(1,3) for i in range(8)])

met = Beat(time=TIME, taps=16, w1=70, w2=70, w3=50).play()
amp = TrigEnv(met, table=env, dur=met['dur'], mul=met['amp'])
fr = TrigXnoiseMidi(met, dist=12, x1=1, x2=.33, scale=0, mrange=(48,85))
frsnap = Snap(fr, choice=SCLS["p1"], scale=1)
osc = LFO(freq=frsnap*cho, sharp=.75, type=3, mul=amp*0.1).mix(2)

met2 = Beat(time=TIME, taps=16, w1=100, w2=40, w3=20).play()
amp2 = TrigEnv(met2, table=env, dur=met2['dur'], mul=met2['amp'])
fr2 = TrigXnoiseMidi(met2, dist=12, x1=1, x2=.5, scale=0, mrange=(36,61))
frsnap2 = Snap(fr2, choice=SCLS["p1"], scale=1)
osc2 = LFO(freq=frsnap2*cho, sharp=.75, type=3, mul=amp2*0.1).mix(2)

rev = WGVerb(osc+osc2, feedback=.7, cutoff=4000, bal=.15).out()

def newbeat(): # Genere de nouvelles sequences rythmiques
    met.new()
    met2.new()

def change(): # actions a la fin de chaque mesure
    print('fin de la mesure')
    if (random.randint(0,100) < 25): # 25% de chance de piger une nouvelle gamme
        gamme = random.choice(['p1', 'p2', 'p3'])
        frsnap.choice = SCLS[gamme]
        frsnap2.choice = SCLS[gamme]
        print("Nouvelle gamme :", gamme)
    if (random.randint(0,100) < 20): # 20% de chance de piger un nouveau rythme
        print("Change de rythme")
        newbeat()

# Appelle la fonction change a chaque fin de mesure
ch = TrigFunc(met['end'], change)

s.gui(locals())
```

scripts/chapitre_08/09_contrepoint.py

# Chapitre 9

# Classes et méthodes

## 9.1 Introduction au concept de classe et d'objets

Le concept de classe est un des éléments les plus importants de la programmation orientée objet (*Object Oriented Programming* ou *OOP*). Un objet est une **instance** créée à partir d'une classe, c'est-à-dire à partir de la définition d'un type particulier d'objet. La définition d'un objet, introduite par le mot-clé **class**, consiste en un regroupement de tous les attributs et méthodes liés à la gestion d'un processus donné. À partir de cette définition, il est possible de créer autant d'objets indépendants qu'il est nécessaire au bon fonctionnement du programme. Ce type de programmation permet de construire des programmes complexes en organisant un ensemble d'objets, ayant chacun leur rôle à jouer, qui interagissent entre eux. Cette approche est bénéfique par le fait que les différents objets peuvent être construits indépendamment les uns des autres sans qu'il n'y ait de risque d'interférence, grâce au concept d'**encapsulation**. En effet, les fonctionnalités internes de l'objet ainsi que les variables qu'il utilise pour effectuer sa tâche sont "enfermées" à l'intérieur de l'objet. Pour y avoir accès de l'extérieur, il faut utiliser des procédures bien définies, ce qui minimise les risques de conflit.

Les concepts suivants débordent un peu le cadre de ce cours, mais il est bon d'en connaître l'existence. Il est possible de créer une classe à partir de la définition d'une autre classe, c'est-à-dire de réutiliser des morceaux de code déjà écrits pour créer une fonctionnalité nouvelle, plus spécifique. Cela est possible grâce aux concepts d'**héritage** et de **polymorphisme**.

— Le mécanisme d'**héritage** permet de construire une classe "enfant" à partir d'une classe "parente". La classe "enfant" hérite ainsi des propriétés de son ancêtre, auxquelles on peut ajouter d'autres fonctionnalités.

— Le **polymorphisme** permet d'attribuer des comportements différents, lors de l'appel d'une fonction par exemple, à des objets dérivant les uns des autres, ou au même objet en fonction d'un certain contexte.

## 9.2 Définition d'une classe simple et de ses attributs

Une classe est créée avec l'instruction **class** suivit du nom que l'on désire lui donner (par convention, les noms de classe débutent toujours par une lettre majuscule). Une classe étant une instruction composée, il est donc nécessaire de terminer la ligne d'en-tête par le symbole 'deux-points' ( : ) et d'indenter tout le code constituant le corps de la classe. Contrairement à la définition d'une fonction, le nom d'une classe n'est pas forcément suivit d'une paire de parenthèses. En fait, celles-ci sont nécessaires seulement lorsque l'on veut désigner une classe parente à la classe que l'on construit, sujet qui ne sera pas couvert dans ce cours. Voici un exemple de définition d'une classe qui nous servira à créer des notes en format Midi :

```
class Note:
    "Instance d'une note Midi"
```

Cette classe, bien qu'inutile pour l'instant, nous permet déjà de créer des objets *Note*. Puisque qu'aucune fonctionnalité n'est encore définie à l'intérieur de la classe, la seule opération pertinente serait d'appeler le contenu de la variable de documentation créée automatiquement par Python !

```
>>> n = Note()
>>> print(n.__doc__)
Instance d'une note Midi
```

Deux détails à noter :

— La création d'un objet se fait par l'appel de la classe suivit d'une paire de parenthèses (nous verrons plus tard que des arguments peuvent être donnés à la création d'un objet). Cet appel retourne un objet que l'on peut affecter à une variable à l'aide du symbole 'égal' ( = ).

— Pour avoir accès aux attributs ou aux méthodes d'un objet, on utilise la syntaxe *objet.attribut* ou *objet.méthode*(). Le point spécifie à l'interpréteur de chercher tel attribut ou telle méthode appartenant à tel objet.

Reprenons la classe *Note* et donnons-lui quelques attributs par défaut, par exemple la hauteur et la vélocité d'une note Midi.

```
class Note:
    "Instance d'une note Midi"
    pitch = 48
    velocity = 127
```

Dès lors, tout objet créé à partir de cette classe possédera deux attributs, *pitch* et *velocity*, avec comme valeurs respectives 48 et 127. Il sera désormais possible de consulter ou de modifier les attributs de chacun des objets créés, indépendamment les uns des autres.

```
>>> n1 = Note()
>>> print(n1.pitch, n1.velocity)
48 127
>>> n2 = Note()
>>> n1.pitch, n1.velocity = 60, 100
>>> print(n1.pitch, n1.velocity)
60 100
>>> print(n2.pitch, n2.velocity)
48 127
```

Nous venons de voir comment définir notre propre classe permettant de créer des objets. La fonctionnalité d'un objet est généralement définie par les méthodes dont la classe est constituée. Dans la prochaine section, nous allons voir les étapes pour créer nos propres méthodes et comment utiliser ces méthodes pour modifier le comportement d'un objet.

## 9.3 Définition d'une méthode

On définit une méthode comme on définit une fonction, avec deux différences cependant :

— La définition d'une méthode doit toujours être encapsulée à l'intérieur d'une classe, c'est-à-dire qu'elle doit faire partie du bloc d'instructions définissant la classe.

— Le premier paramètre utilisé par une méthode doit toujours être une référence d'instance. Par convention, le mot réservé **self** est utilisé pour désigner l'instance. Cela veut dire qu'il doit toujours y avoir au moins un paramètre entre les parenthèses lors de la définition d'une méthode.

```
class Note:
    "Instance d'une note Midi"
    def maFonction(self):
        print('Je suis une methode de la classe Note')
        print(self)
```

```
>>> n = Note()
>>> n.maFonction()
Je suis une methode de la classe Note
<__builtin__.Note instance at 0x1c0f1f30>
```

Lors de l'appel de la fonction *maFonction*, on constate que **self** correspond bien à une instance de la classe *Note*.

**À noter :**

Bien que la définition de la méthode *maFonction* comporte un argument entre les parenthèses, l'appel se fait sans argument. Ceci est du au fait que lors de l'appel d'une méthode avec la syntaxe *objet.méthode*(), l'objet est lui-même automatiquement donné comme premier argument

à la méthode, d'où la définition avec **self** en argument. Si la méthode nécessite des arguments particuliers, ils seront ajoutés dans la définition à la suite de l'argument **self**. L'appel se fera de façon standard, comme pour une fonction, toujours en ignorant le premier argument (**self**).

```
class Note:
    "Instance d'une note Midi"
    def fonctionAvecArgs(self, x, y):
        print(self)
        print(x, y)
```

```
>>> n = Note()
>>> n.fonctionAvecArgs(10, 20)
<__builtin__.Note instance at 0x1c0f1f30>
10 20
```

Il existe une méthode, qui est automatiquement appelée à la création d'un objet, permettant d'initialiser des variables d'instances, c'est la méthode "constructeur".

## 9.4 La méthode "constructeur"

La méthode "constructeur" est une méthode particulière qui est appelée automatiquement lors de la création d'un objet. Cette méthode doit obligatoirement s'appeler __**init**__ (deux caractères 'souligné', le mot **init**, puis deux autres caractères 'souligné').

```
class Note:
    "Instance d'une note Midi"
    def __init__(self):
        self.pitch = 48
        self.velocity = 127

    def getNote(self):
        "Retourne la note Midi"
        return (self.pitch, self.velocity)
```

Dans le code ci-dessus, on constate que les attributs sont maintenant initialisés à l'intérieur de la méthode "constructeur". On constate aussi que le nom de ces attributs est précédé du mot **self** et d'un point. Cela permet de définir des **variables d'instance**, c'est à dire des variables qui appartiennent à l'objet créé et qui seront accessibles n'importe où à l'intérieur de l'objet. Ainsi, la fonction *getNote* peut retourner les valeurs de hauteur et de vélocité puisqu'elle a accès aux variables d'instance de l'objet. Si plusieurs instances sont créées à partir d'une même classe, les variables d'instance sont indépendantes d'un objet à l'autre. Il est donc possible de modifier la variable *self.pitch* sans risque d'interférence avec la hauteur de la note des autres instances de la classe.

Donnons maintenant une fonctionnalité propre à notre classe. En plus de retourner la note Midi telle quelle, nous allons définir deux autres fonctions. La première effectuera la conversion

de la hauteur Midi en cycles par seconde (Hertz) tandis que la deuxième transposera la vélocité en amplitude normalisée entre 0 et 1.

```
class Note:
    "Instance d'une note Midi"
    def __init__(self):
        self.pitch = 48
        self.velocity = 127

    def getNote(self):
        "Retourne la note Midi"
        return (self.pitch, self.velocity)

    def mtof(self):
        "Retourne la frequence en Hertz de la note"
        return 8.175798 * pow(1.0594633, self.pitch)

    def vtoa(self):
        "Retourne l'amplitude de la note entre 0 et 1"
        return self.velocity / 127.
```

Dans le cas où l'on désirerait spécifier les valeurs de hauteur et de vélocité de la note au moment de la création de l'objet, il suffirait de donner ces valeurs en paramètres à la création et de les récupérer dans la méthode "constructeur", en plaçant des arguments à la suite de la variable d'instance.

```
>>> class Note:
...     "Instance d'une note Midi"
...     def __init__(self, pit, vel):
...         self.pitch = pit
...         self.velocity = vel
...
>>> n = Note(36, 117)
```

Voici un petit exemple d'utilisation de la classe "Note" afin de générer une mélodie.

```
1  from pyo import *
2  import random
3
4  class Note:
5      "Conteneur pour note Midi"
6      def __init__(self, pit, vel):
7          self.pitch = pit
8          self.velocity = vel
9
10     def getNote(self):
11         "Retourne la note Midi"
12         return (self.pitch, self.velocity)
13
14     def mtof(self):
15         "Retourne la frequence en Hertz de la note"
16         return 8.175798 * pow(1.0594633, self.pitch)
17
18     def vtoa(self):
```

```
        "Retourne l'amplitude de la note entre 0 et 1"
        return self.velocity / 127.

num_notes = 200
notes_list = []

for i in range(num_notes):
    # Pige aleatoire d'une valeur paire entre 48 et 84
    pit = random.randrange(48, 85, 2)
    vel = random.randint(10, 30)
    notes_list.append(Note(pit, vel))

s = Server().boot()

# Initialisation de listes vides pour garder les references aux objets audio
fades = []
oscs = []

### Initialisation des notes ###
# args : delay = delai avant d'activer l'objet, dur = temps d'activation de
# l'objet (la methode stop() est appelee automatiquement)
for i in range(num_notes):
    # Recupere un objet Note
    note = notes_list[i]
    # Temps de depart de la note
    start = i * .125
    # Enveloppe d'amplitude. L'amplitude est recuperee par la methode vtoa()
    f = Fader(fadein=.01, fadeout=.99, dur=1,
              mul=note.vtoa()).play(delay=start, dur=1)
    # Oscillateur. La frequence est recuperee par la methode mtof()
    osc = SineLoop(freq=note.mtof(), feedback=.05,
                   mul=f).out(i%2, delay=start, dur=1)
    # Ajoute les objets dans les listes
    fades.append(f)
    oscs.append(osc)

s.gui(locals())
```

scripts/chapitre_09/01_classe_note_1.py

Cette technique nous permet de gérer la génération des notes en pur Python, environnement très puissant en ce qui concerne l'algorithmie. Dans le script suivant, nous allons remplacer la boucle qui pige des valeurs aléatoires pour la hauteur et la vélocité par une boucle un peu plus raffinée. Au lieu de piger des hauteurs dans le hasard le plus complet, nous allons créer une marche aléatoire en ajoutant une petite valeur à la hauteur précédente, minimisant ainsi les sauts dans la mélodie. Pour la vélocité, nous allons imposer des accents en donnant systématiquement une vélocité élevée à la première note de chaque groupe de 4. Le script suivant remplace la boucle de génération de notes par notre nouvel algorithme.

```
from pyo import *
import random

class Note:
    "Conteneur pour note Midi"
    def __init__(self, pit, vel):
        self.pitch = pit
        self.velocity = vel

    def getNote(self):
        "Retourne la note Midi"
        return (self.pitch, self.velocity)

    def mtof(self):
        "Retourne la frequence en Hertz de la note"
        return 8.175798 * pow(1.0594633, self.pitch)

    def vtoa(self):
        "Retourne l'amplitude de la note entre 0 et 1"
        return self.velocity / 127.

num_notes = 200
notes_list = []

pit = 64 # Initialisation de la hauteur de depart
for i in range(num_notes):
    # Marche aleatoire. Ajoute une valeur paire entre -4 et 5 au dernier pitch
    pit = random.randrange(-4, 5, 2) + pit
    # Impose des limites
    if pit < 48:
        pit = 48
    elif pit > 84:
        pit = 84
    if (i % 4) == 0:
        vel = 40 # Temps forts
    else:
        vel = random.randint(8, 25) # Temps faibles
    notes_list.append(Note(pit, vel))

s = Server().boot()

fades, oscs = [], []
for i in range(num_notes):
    note = notes_list[i]
    start = i * .125
    f = Fader(fadein=.01, fadeout=.99, dur=1, mul=note.vtoa()).play(delay=start,
        dur=1)
    osc = SineLoop(freq=note.mtof(), feedback=.05, mul=f).out(i%2, delay=start,
        dur=1)
    fades.append(f)
    oscs.append(osc)

s.gui(locals())
```

scripts/chapitre_09/02_classe_note_2.py

## 9.5 Exemple de classe dans un fichier externe

Le principe de se créer une librairie personnelle consiste à écrire un module contenant une collection de classes que l'on peut importer à volonté dans nos scripts. Voici un exemple de module contenant une classe générant des particules de son.

```
from pyo import *

class Particule:
    def __init__(self, input, time=.125, q=20,
                 maxdur=.2, frange=[350,10000], poly=12):
        # Enveloppe d'amplitude
        self.table = LinTable([(0,0),(100,1),(500,.3),(8191,0)])
        # InputFader permet de faire des crossfades sur le son source
        self.input = InputFader(input)
        # Tous les parametres variables sont convertis en audio
        self.time = Sig(time)
        self.q = Sig(q)
        self.maxdur = Sig(maxdur)

        # Generation de trigs
        self.metro = Metro(self.time, poly=poly).play()
        # Pige de valeurs de frequence et de duree
        self.freq = TrigRand(self.metro, min=frange[0], max=frange[1])
        self.dur = TrigRand(self.metro, min=.01, max=self.maxdur)
        # Lecture de l'enveloppe d'amplitude
        self.amp = TrigEnv(self.metro, table=self.table, dur=self.dur)
        # Particule filtree
        self.filtre = Biquadx(self.input, freq=self.freq, q=self.q,
                              type=2, stages=2, mul=self.amp)

    def out(self):
        "Envoie le son aux haut-parleurs et retourne l'objet lui-meme."
        self.filtre.out()
        return self

    def getOut(self):
        """ Retourne l'objet audio qui genere le signal de sortie de la
        classe afin de pouvoir l'inclure dans une chaine de traitement."""
        return self.filtre

    def setInput(self, x, fadetime=.05):
        self.input.setInput(x, fadetime)

    def setTime(self, x):
        self.time.value = x

    def setQ(self, x):
        self.q.value = x

    def setMaxDur(self, x):
        self.maxdur.value = x
```

scripts/chapitre_09/malibrairie.py

Ensuite, il suffit d'importer "malibrairie" dans un script pour avoir accès aux classes qui y sont définies. Dans un second fichier (situé dans le même dossier que "malibrairie.py" afin d'éviter les problèmes), on importe et on utilise la classe "Particule" :

```
#!/usr/bin/env python
# encoding: utf-8
from pyo import *
from malibrairie import Particule
import random

s = Server().boot()

### Differentes sources sonores ###
a = Noise(.3)
a1 = SineLoop(freq=50, feedback=.2, mul=1)
a2 = FM(carrier=100, ratio=5.67, index=20, mul=.3)

### 5 exemples d'utilisation de la classe Particule ###
which = 0
if which == 0:
    b = Particule(a, time=.125, q=2, maxdur=.5).out()
elif which == 1:
    qlfo = Sine(.1, mul=5, add=6)
    b = Particule(a, time=.125, q=qlfo, maxdur=.25).out()
elif which == 2:
    tlfo = Sine(.15, mul=.1, add=.15)
    qlfo = Sine(.1, mul=5, add=6)
    b = Particule(a, time=tlfo, q=qlfo, maxdur=.25).out()
elif which == 3:
    b = Particule(input=a, time=.25, q=10, maxdur=.5, frange=[350,1000]).out()
    c = Particule(a, time=.125, q=5, maxdur=.25, frange=[2000,10000]).out()
elif which == 4:
    tlfo = Sine(.15, mul=.1, add=.15)
    qlfo = Sine(.1, mul=5, add=6)
    b = Particule(a, time=tlfo, q=qlfo, maxdur=.25)
    d = Delay(b.getOut().mix(2), delay=.5, feedback=.75)
    c = Freeverb(d, size=.8, damp=.9, bal=.4).out()

def change():
    y = random.randint(0,2)
    if y == 0:
        b.setInput(a, 2)
    elif y == 1:
        b.setInput(a1, 2)
    elif y == 2:
        b.setInput(a2, 2)

# "pat.play()" active le changement de source automatique
pat = Pattern(change, 5)

s.gui(locals())
```

scripts/chapitre_09/03_classe_particule.py

# Chapitre 10

# Contrôleurs externes

La manipulation des processus audio via des contrôleurs externes est très efficace pour produire des variations de paramètres aux contours naturels. Deux protocoles de communication seront explorés dans ce chapitre. Nous allons tout d'abord nous attarder au protocole Midi qui permettra la création de synthétiseurs virtuels simples et efficaces. Puis, nous élaborerons sur le protocole OSC (*Open Sound Control*) qui offre un contrôle du data plus généralisé que le protocole Midi.

## 10.1 Configuration Midi

Pour utiliser une interface Midi dans un script python, il faut d'abord s'assurer que le serveur audio (c'est l'objet *Server* qui se charge des entrées/sorties Midi) reçoit bien le data de l'interface désirée. Deux fonctions sont disponibles pour tester la configuration Midi en entrée :

— *pm_list_devices()* affiche l'ensemble des interfaces connectées au système.

— *pm_get_default_input()* retourne le numéro de l'interface par défaut.

```
>>> pm_list_devices()
MIDI devices:
0: IN, name: IAC Driver Bus 1, interface: CoreMIDI
1: IN, name: Keystation Port 1, interface: CoreMIDI
2: OUT, name: IAC Driver Bus 1, interface: CoreMIDI
3: OUT, name: Keystation Port 1, interface: CoreMIDI

>>> pm_get_default_input()
0
```

Si l'interface Midi par défaut n'est pas l'interface désirée, il suffit de donner le numéro de l'interface à la méthode *setMidiInputDevice()* de l'objet **Server** avant d'appeler la méthode *boot()*. Avec la configuration ci-dessus, pour utiliser le clavier Midi **Keystation**, on doit spécifier l'interface numéro 1 :

```
s = Server()
s.setMidiInputDevice(1)
s.boot()
```

Petit truc : Si la méthode *setMidiInputDevice()* reçoit un nombre plus élevé que le plus grand numéro retourné par la fonction *pm_list_devices()*, toutes les interfaces Midi disponibles seront accesibles dans le programme.

## 10.2 Synthétiseur Midi

Ce script implémente une classe générant un son de synthèse sur réception de notes Midi.

```
from pyo import *

class Synth:
    def __init__(self, transpo=1):
        # Facteur de transposition
        self.transpo = Sig(transpo)
        # Recoit les notes Midi et les assignes sur 10 voix de polyphonie
        self.note = Notein(poly=10, scale=1, first=0, last=127)
        # Traitement des valeurs de hauteur et d'amplitude des notes Midi
        self.pit = self.note['pitch'] * self.transpo
        self.amp = MidiAdsr(self.note['velocity'], attack=0.001,
                            decay=.1, sustain=.7, release=2, mul=.1)
        # Chorus de 4 oscillateurs, mixes en mono pour eviter
        # l'alternance panoramique des 10 voix de polyphonie
        self.osc1 = SineLoop(self.pit, feedback=0.12, mul=self.amp).mix(1)
        self.osc2 = SineLoop(self.pit*.997, feedback=0.12, mul=self.amp).mix(1)
        self.osc3 = SineLoop(self.pit*.994, feedback=0.12, mul=self.amp).mix(1)
        self.osc4 = SineLoop(self.pit*1.002, feedback=0.12, mul=self.amp).mix(1)
        # Mixage stereo
        self.mix = (self.osc1+self.osc2+self.osc3+self.osc4).mix(2)

    def out(self):
        "Active l'envoi du signal aux sorties et retourne l'objet lui-meme."
        self.mix.out()
        return self

    def sig(self):
        "Retourne l'objet qui produit le signal final de la chaine audio."
        return self.mix

# Configuration audio/midi
idev = pm_get_default_input()
s = Server()
s.setMidiInputDevice(idev) # Changer idev en fonction de votre configuration
s.boot()

a1, a2 = Synth(), Synth(.5) # octave reel et octave inferieur

# La Somme des signaux des synths passe dans une reverberation
rev = WGVerb(a1.sig()+a2.sig(), feedback=.8, cutoff=3500, bal=.3).out()

```

```
s.gui(locals())
```

scripts/chapitre_10/01_midi_synth.py

Deux nouveaux objets sont introduit ici, Notein et MidiAdsr. Le premier reçoit les notes du clavier Midi et génère deux *streams* audio distincts par note, un pour la hauteur (*self.note['pitch']*) et un pour l'amplitude (*self.note['velocity']*). Le second génère une enveloppe **ADSR** classique en fonction de la vélocité de la note. Voici une version légèrement bonifiée de notre programme !

```
from pyo import *
from random import uniform

class Synth:
    def __init__(self, transpo=1):
        self.transpo = Sig(transpo)
        self.note = Notein(poly=8, scale=1, first=0, last=127)
        self.pit = self.note['pitch'] * self.transpo
        self.amp = MidiAdsr(self.note['velocity'], attack=0.001,
                            decay=.1, sustain=.7, release=1, mul=.3)

        self.osc1 = LFO(freq=self.pit, sharp=0.25, mul=self.amp).mix(1)
        self.osc2 = LFO(freq=self.pit*0.997, sharp=0.25, mul=self.amp).mix(1)
        self.osc3 = LFO(freq=self.pit*1.004, sharp=0.25, mul=self.amp).mix(1)
        self.osc4 = LFO(freq=self.pit*1.009, sharp=0.25, mul=self.amp).mix(1)

        # Mix stereo (osc1 et osc3 a gauche, osc2 et osc4 a droite)
        self.mix = Mix([self.osc1+self.osc3, self.osc2+self.osc4], voices=2)

        # Distortion avec LFO sur le drive
        self.lfo = Sine(freq=uniform(.2,.4), mul=0.45, add=0.5)
        self.disto = Disto(self.mix, drive=self.lfo, slope=0.95, mul=.2)

    def out(self):
        self.disto.out()
        return self

    def sig(self):
        return self.disto

# Configuration audio/midi
idev = pm_get_default_input()
s = Server()
s.setMidiInputDevice(idev)
s.boot()

# roue de modulation = amplitude du vibrato
ctl = Midictl(1, minscale=0, maxscale=.2)
bend = Bendin(brange=2, scale=1) # Pitch bend
lf = Sine(freq=5, mul=ctl, add=1) # Vibrato

a1 = Synth(lf * bend)
comp = Compress(a1.sig(), thresh=-20, ratio=6)
rev = WGVerb(comp, feedback=.8, cutoff=3500, bal=.3).out()

```

```
s.gui(locals())
```

scripts/chapitre_10/02_midi_synth_2.py

Ce programme présente deux nouveaux objets de la librairie Midi, Midictl, pour récupérer les valeurs d'un contrôleur continu, et Bendin, pour récupérer les valeurs du contrôleur *Pitch Bend*.

## 10.3 Générateur de boucles audio contrôlable en Midi

La classe suivante lit un ou plusieurs fichiers sons en boucle et offre le contrôle sur les transpositions via le clavier Midi.

```
from pyo import *

class SndSynth:
    def __init__(self, path, first=0, last=127):
        # Place le son en memoire
        self.table = SndTable(path)
        # scale=2 convertit les notes midi en facteurs de transposition
        self.notein = Notein(poly=3, scale=2, first=first, last=last)
        self.transpo = self.notein['pitch'] * self.table.getRate()
        self.amp = Port(self.notein['velocity'], risetime=0.001, mul=.5)

        # Osc -> lit une table en boucle
        self.osc = Osc(self.table, freq=self.transpo, mul=self.amp).mix(1)
        self.mix = self.osc.mix(2)

    def out(self):
        self.mix.out()
        return self

    def sig(self):
        return self.mix

idev = pm_get_default_input()
s = Server()
s.setMidiInputDevice(idev)
s.boot()

# Clavier Midi separe en regions par tierce mineure
# autour des notes 60, 63, 66, 69, 72, 75
li = []
for i in range(1, 7):
    filename = 'snd_%i.aif' % i
    centralkey = 57 + i * 3
    li.append(SndSynth(filename, first=centralkey-1, last=centralkey+1).out())

s.gui(locals())
```

scripts/chapitre_10/03_midi_synth_snd.py

## 10.4 Simili-échantillonneur

Cette classe permet le déclenchement et la transposition de lectures de sons via le clavier Midi. L'objet **Notein** envoit un *trigger* sur réception d'une note à vélocité positive via le *stream* audio « trigon » (*self.notein["trigon"]*). Ce dernier sert à activer les lectures de table.

```
from pyo import *

class SndSynth:
    def __init__(self, path, first=0, last=127):
        self.table = SndTable(path)
        self.notein = Notein(poly=10, scale=2, first=first, last=last)
        # Duree de lecture de la table = duree_table / facteur_transpo
        self.dur = self.table.getDur() / self.notein['pitch']
        self.amp = Port(self.notein['velocity'], risetime=0.001, mul=.5)

        # Notein["trigon"] envoie un trig sur reception d'un noteon.
        self.snd = TrigEnv(self.notein["trigon"], table=self.table,
                           dur=self.dur, mul=self.amp).mix(1)
        self.mix = self.snd.mix(2)

    def out(self):
        self.mix.out()
        return self

    def sig(self):
        return self.mix

# Affiche la liste des interfaces midi
pm_list_devices()

# Remplacer la valeur en idev par l'interface desiree
idev = pm_get_default_input()

s = Server()
s.setMidiInputDevice(99)
s.boot()

# Clavier Midi separe en regions par octave entre les DOs
li = []
for i in range(1, 7):
    filename = 'snd_%i.aif' % i
    firstkey = i * 12 + 24
    li.append(SndSynth(filename, first=firstkey, last=firstkey+11).out())

s.gui(locals())
```

scripts/chapitre_10/04_midi_synth_snd_2.py

## 10.5 Granulateur simple

Est présenté ici un granulateur de fichiers sons qui servira de prétexte au contrôle de paramètre via le protocole OSC (*Open Sound Control*).

```
from pyo import *

s = Server().boot()

env = HannTable()
table = SndTable("Beethoven.aif")
table.view()

# exemple 0 -> changement de vitesse sans toucher a la hauteur
# exemple 1 -> position de lecture aleatoire
ex = 0
if ex == 0:
    pos = Phasor(freq=table.getRate() * 0.25, mul=table.getSize())
elif ex == 1:
    pos = Randi(min=0, max=table.getSize(), freq=2)

pit = [1., 1.002]
dur = Noise(mul=.002, add=.1)   # legeres variations de la duree des grains

grain = Granulator( table=table,    # table contenant les echantillons
                    env=env,        # enveloppe des grains
                    pitch=pit,      # pitch global des grains
                    pos=pos,        # position de lecture, en echantillons
                    dur=dur,        # duree des grains en secondes
                    grains=32,      # nombre de grains
                    basedur=.1,     # duree de reference pour les grains
                                    # (si dur == basedur, pas de transpo)
                    mul=.1).out()

s.gui(locals())
```

scripts/chapitre_10/05_granulateur.py

## 10.6 Contrôle du granulateur via le protocole OSC

Pour l'exemple suivant, nous utiliserons deux scripts python. Le premier reprend le granulateur de l'exemple précédent et modifie les contrôles pour recevoir des données via le protocole OSC. Le second script contrôle la position de lecture ainsi que la durée des grains du granulateur en propageant les données sur le numéro de *port* qu'écoute le premier script. Les données de contrôle, dans un envoie OSC, sont identifiées à l'aide d'un système d'adresses. Les adresses, spécifiées en format *string*, utilisent la syntaxe des *paths Unix*, avec la barre oblique ( / ) comme séparateur. Exemple : '/data/freq', '/data/amp', etc.

```python
from pyo import *

s = Server().boot()

table = SndTable("Beethoven.aif")
env = HannTable()

# Ecoute le port 9000 aux destinations '/pos' et '/rand'
rec = OscReceive(port=9000, address=['/pos', '/rand'])

# rec['/pos'] -> position de lecture
# rec['/rand'] -> random pour la duree des grains
pos = Port(rec['/pos'], risetime=.05, falltime=.05, mul=table.getSize())

pit = [1., 1.002]

# legeres variations de la duree des grains
dur = Noise(mul=Clip(rec['/rand'], min=0, max=.09), add=.1)

grain = Granulator( table=table,    # table contenant les echantillons
                    env=env,        # enveloppe des grains
                    pitch=pit,      # pitch global des grains
                    pos=pos,        # position de lecture, en echantillons
                    dur=dur,        # duree des grains en secondes
                    grains=32,      # nombre de grains
                    basedur=.1,     # duree de reference pour les grains
                    mul=.1).out()

s.gui(locals())
```

scripts/chapitre_10/06_osc_receive.py

```python
from pyo import *

s = Server().boot()

# Un lfo qui controle la position de lecture normalisee entre 0 et 1
lfo = Sine(.1, mul=.5, add=.5)

# Controle manuel sur la transposition par grain
rnd = Sig(0)
rnd.ctrl(title="Transposition par grain")

# Envoie les controles sur le port 9000 aux destinations '/pos' et '/rand'
# host est l'adresse IP de la machine ciblee. 127.0.0.1 = machine locale
send = OscSend([lfo, Pow(rnd, 6)], port=9000, address=['/pos','/rand'], host='
    127.0.0.1')

s.gui(locals())
```

scripts/chapitre_10/07_osc_send.py

# Chapitre 11

# Cecilia5 API - Écriture de module

## 11.1 Introduction

### 11.1.1 Qu'est-ce qu'un module Cecilia ?

Un module Cecilia est un fichier python (portant l'extension ".c5", propre à l'application Cecilia) contenant un classe *Module*, à l'intérieur de laquelle la chaîne de traitements audio est développée, et une liste *Interface*, permettant au logiciel de construire tous les contrôles graphiques nécessaires au bon fonctionnement du module. Le fichier peut alors être chargé par l'application afin d'appliquer la chaîne de traitements sur différents signaux, qu'ils proviennent de fichiers sonores ou de l'entrée microphone. Les processus utilisés pour manipuler le signal audio doivent être écrits avec le module dédié au traitement de signal *pyo*.

### 11.1.2 Documentation

L'API (*Application Programming Interface*) de Cecilia peut être consulté à même l'application, via la commande *Show API Documentation*, sous le menu *Help*. Il se divise en deux parties : d'une part, il y a la documentation nécessaire au développement du module lui-même, c'est-à-dire la classe où est décrite la chaîne de traitements à appliquer au signal audio. Ensuite, les différents éléments d'interface graphique (*widgets*) disponibles sont présentés. Le présent chapitre suit la même structure.

## 11.2 BaseModule

Tout module Cecilia doit impérativement comporter une classe nommée "Module", dans laquelle sera développée la chaîne de traitements sonores. Cette classe, pour fonctionner correctement dans l'environnement Cecilia, doit hériter de la classe "BaseModule", définie à même le code source de Cecilia. C'est la classe "BaseModule" qui se charge de créer, à l'interne, les liens entre l'interface graphique, les communications externes (MIDI, OSC) et le processus audio lui-même.

### 11.2.1 Initialisation

Un module doit être déclaré comme ceci :

```
class Module(BaseModule):
    def __init__(self):
        BaseModule.__init__(self)
        # Here comes the processing chain...
```

Comme le fichier sera exécuté à l'intérieur de l'environnement Cecilia, il n'est pas nécessaire d'importer *pyo* ou la classe *BaseModule*, à laquelle on fait référence en tant que classe parente. Ces composantes étant déjà importées par Cecilia, elles seront disponibles à l'exécution du fichier.

Voici le détail de chacune des lignes de l'extrait précédent :

```
class Module(BaseModule):
```

Ligne d'en-tête de la classe. Le nom de la classe doit obligatoirement être "Module". Elle hérite bien de la classe "BaseModule", définie dans le code de Cecilia.

```
    def __init__(self):
```

Méthode "constructeur" de la classe. Aucun argument ne doit être défini (à l'exception, bien sur, de *self*).

```
        BaseModule.__init__(self)
```

Initialisation (appel de la méthode "constructeur") de la classe parente. Cette étape est indispensable afin que les liens entre l'interface et le processus audio soient créés.

```
        # Here comes the processing chain...
```

Lorsque tout est correctement initialisé, on peut développer notre chaîne de traitements de signal !

### 11.2.2 Sortie audio de la classe

Afin d'acheminer le signal audio du processus à l'application, "self.out" doit absolument être le nom de variable de l'objet à la toute fin de la chaîne de traitement. Cecilia récupère cette variable pour faire suivre le signal audio vers la section Post-Processing et ultimement, à la sortie audio. Un example typique de déclaration de la variable "self.out" ressemblerait à ceci :

```
self.out = Interp(self.snd, self.dsp, self.drywet, mul=self.env)
```

L'objet *Interp* permet d'interpoler entre le signal original ("self.snd") et le signal modifié ("self.dsp") en fonction d'un potentiomètre de l'interface ("self.drywet"). Une enveloppe d'amplitude générale ("self.env"), définie comme une ligne de graphe, est appliquée au signal final.

### 11.2.3 Documentation du module

Les informations pertinentes à une bonne compréhension du comportement d'un module peuvent être données dans le *__doc__ string* de la classe "Module". L'utilisateur de Cecilia pourra afficher la documentation du module à l'écran via la commande *Show module info* du menu *Help* de l'application.

### 11.2.4 Attributs publics de la classe *BaseModule*

Ces variables, définies à l'initialisation de la classe "BaseModule", contiennent des informations sur la configuration courante de Cecilia et peuvent être utilisées à tout moment dans la gestion du processus audio.

- **self.sr** : Fréquence d'échantillonnage définie à même l'interface de Cecilia.
- **self.nchnls** : Nombre de canaux audio défini dans l'interface.
- **self.totalTime** : Durée totale de la performance, définie dans l'interface.
- **self.number_of_voices** : Nombre de voix de polyphonie provenant d'un *cpoly* (voir section sur la définition de l'interface).
- **self.polyphony_spread** : Liste des facteurs de transposition provenant d'un *cpoly*.
- **self.polyphony_scaling** : Valeur servant à pondérer l'amplitude du processus en fonction du nombre de voix de polyphonie (*cpoly*).

### 11.2.5 Méthodes publiques de la classe *BaseModule*

Les méthodes suivantes sont définies à l'intérieur de la classe "BaseModule" et peuvent donc être utilisées dans la composition de la chaîne de traitements du module.

- **self.addFilein(name)** : Crée et retourne un objet *SndTable* en fonction du nom donné à un *cfilein*.
- **self.addSampler(name, pitch, amp)** : Crée un *sampler/looper* en fonction du nom donné à un *csampler*. La fonction retourne le signal audio du *sampler*.
- **self.getSamplerDur(name)** : Retourne la durée, en secondes, du son chargé dans un *sampler*.
- **self.duplicate(seq, num)** : Duplique les éléments d'une liste *seq* selon le nombre d'itérations demandées à l'argument *num*. Retourne la nouvelle liste.
- **self.setGlobalSeed(x)** : Assigne une valeur fixe à la racine des générateurs aléatoires, via le serveur *pyo*. Cette fonction permet d'obtenir les mêmes séquences algorithmiques à chaque performance.

## 11.3 Éléments d'interface graphique

Le fichier Cecilia (.c5), en plus de la classe audio, doit spécifier la liste des contrôles graphiques nécessaires au bon fonctionnement du module. Cette liste doit impérativement porter le nom "Interface" et être constituée d'appels de fonctions propres à l'environnement Cecilia. Voici un exemple où est déclaré un lecteur de son, une ligne de graphe, deux potentiomètres et un contrôle de la polyphonie :

```
Interface = [
    csampler(name="snd"),
    cgraph(name="env", label="Overall Amp", func=[(0,1),(1,1)], col="blue1"),
    cslider(name="freq", label="Filter Freq", min=20, max=20000, init=1000,
            rel="log", unit="Hz", col="green1"),
    cslider(name="q", label="Filter Q", min=0.5, max=25, init=1, rel="log",
            unit="x", col="green2"),
    cpoly()
]
```

Voici la liste des fonctions disponibles et ainsi que la description de l'élément d'interface qu'elles permettent de créer.

### 11.3.1 cfilein

La fonction *cfilein* crée un menu déroulant permettant à l'utilisateur de charger un son dans une table (espace mémoire) et d'utiliser celle-ci dans le module de traitement. Lorsque l'utilisateur choisi un son sur le disque, tout le dossier est parcouru et les fichiers sonores qui s'y trouvent peupleront le menu déroulant, permettant ainsi un accès rapide aux différents sons. Un clic sur le menu ouvre le dialogue standard pour parcourir la hiérarchie sur le disque et un clic sur la flèche à droite ouvre le menu déroulant.

Plusieurs *cfilein* peuvent être définis pour un même module, tant qu'ils portent un nom différents. Ils apparaîtront un à la suite de l'autre, selon l'ordre donné dans la liste, dans le panneau "Input" de l'interface. Le *cfilein* se charge de créer une table dont le nombre de canaux est conséquent avec le nombre de canaux de Cecilia au moment où la performance est lancée, et ce, peut importe le nombre de canaux du fichier son.

Pour récupérer la table ainsi créée dans la classe "Module", on utilise la méthode *addFilein(name)* définie dans la classe "BaseModule". On donne le nom du *cfilein* en argument à la méthode.

#### Exemple

```
class Module(BaseModule):
    def __init__(self):
        BaseModule.__init__(self)
        self.table = self.addFilein("source")
        self.out = Osc(self.table, freq=self.table.getRate(), mul=self.env*.2)

Interface = [
        cfilein(name="source", label="Source Audio"),
        cgraph(name="env", label="Overall Amp", func=[(0,1),(1,1)], col="blue1"),
]
```

#### Déclaration

| **cfilein**(*name*="filein", *label*="Audio") |
|---|

### Arguments

- **name** : {str} Chaîne de caractères permettant la communication entre l'élément d'interface et le module de traitement. La même chaîne de caractères doit être donnée à une méthode *addFilein*.
- **label** : {str} Chaîne de caractères descriptive qui apparaît juste au-dessus du menu déroulant dans l'interface graphique.

## 11.3.2 csampler

La fonction *csampler* crée un menu déroulant permettant à l'utilisateur de charger un son dans un module de lecture bouclée et d'utiliser le signal de celui-ci comme source audio dans le module de traitement. Lorsque l'utilisateur choisi un son sur le disque, tout le dossier est parcouru et les fichiers sonores qui s'y trouvent peupleront le menu déroulant, permettant ainsi un accès rapide aux différents sons. Un clic sur le menu ouvre le dialogue standard pour parcourir la hiérarchie sur le disque et un clic sur la flèche à droite ouvre le menu déroulant.

Plusieurs *csampler* peuvent être définis pour un même module, tant qu'ils portent un nom différents. Ils apparaîtront un à la suite de l'autre, selon l'ordre donné dans la liste, dans le panneau "Input" de l'interface. Le *csampler* se charge de créer un signal dont le nombre de canaux est conséquent avec le nombre de canaux de Cecilia au moment où la performance est lancée, et ce, peut importe le nombre de canaux du fichier son.

À l'aide du petit triangle de la boîte à outils (à droite du menu déroulant), l'utilisateur peut ouvrir la fenêtre de contrôle du lecture. Il aura alors accès à divers paramètres, notamment, le point de départ et la durée de la boucle, la transposition, l'amplitude, etc.

Pour créer le lecteur en boucle et récupérer le signal dans la classe "Module", on utilise la méthode *addSampler(name, pitch=1, amp=1)* définie dans la classe "BaseModule". On donne le nom du *csampler* désiré à l'argument "name" de la méthode. Optionnellement, il est possible d'assigner des variables aux arguments "pitch" et "amp", afin de contrôler, respectivement, la hauteur et l'amplitude de la lecture. Ex. :

### Exemple

```
class Module(BaseModule):
    def __init__(self):
        BaseModule.__init__(self)
        self.snd = self.addSampler("snd")
        self.out = Mix(self.snd, voices=self.nchnls, mul=self.env)

Interface = [
    csampler(name="snd"),
    cgraph(name="env", label="Overall Amplitude", func=[(0,1),(1,1)], col="blue"),
]
```

### Déclaration

**csampler**(*name*="sampler", *label*="Audio")

**Arguments**

- **name** : {str} Chaîne de caractères permettant la communication entre l'élément d'interface et le module de traitement. La même chaîne de caractères doit être donnée à une méthode *addSampler*.
- **label** : {str} Chaîne de caractères descriptive qui apparaît juste au-dessus du menu déroulant dans l'interface graphique.

### 11.3.3 cpoly

La fonction *cpoly* permet une gestion simplifiée de la polyphonie dans un module. Elle met en place deux menus déroulants dans la section des menus et commutateurs (section inférieure gauche). Le premier permet de sélectionner le nombre de voix qui seront jouées en simultanées tandis que le second permet de configurer la distribution des facteurs transpositions (*phasing*, chorus, désaccordés ou un des nombreux accords fournis).

La classe "BaseModule" génère trois variables en lien avec les valeurs spécifiées par le *cpoly*. Elles peuvent être utilisées pour ajuster le nombre d'itérations des processus sonores présents dans le module. Si le module utilise un *csampler* comme source audio, la polyphonie est automatiquement tenu en compte à même la lecture du son, il n'y a donc rien à ajouter. Les variables créées automatiquement sont :

- **self.number_of_voices** : {int} Nombre de voix de polyphonie sélectionné.
- **self.polyphony_spread** : {list} Liste des facteurs de transposition définis en fonction de la distribution choisie.
- **self.polyphony_scaling** : {float} Facteur d'amplitude servant à ajuster le gain du signal en fonction du nombre de voix de polyphonie.

À noter : On ne peut déclarer plus d'un *cpoly* par module.

**Exemple**

```
class Module(BaseModule):
    def __init__(self):
        BaseModule.__init__(self)
        # Duplique les transpositions en fonction du nombre de canaux.
        transXchnls = self.duplicate(self.polyphony_spread, self.nchnls)
        # Frequence du slider * facteurs de transpo. Ajustement de l'amplitude.
        self.saw = LFO(freq=self.freq*transXchnls, mul=self.polyphony_scaling)
        # Mix les signaux en fonction du nombre de canaux.
        self.out = Mix(self.saw, voices=self.nchnls, mul=self.env*0.1)

Interface = [
    cgraph(name="env",label="Overall Amplitude",func=[(0,1),(1,1)],col="blue1"),
    cslider(name="freq",label="Base Freq",max=1000,init=150,rel="log",col="red1"),
    cpoly()
]
```

### Déclaration

**cpoly**(*name*="poly", *label*="Polyphony", *min*=1, *max*=10, *init*=1)

### Arguments

- **name** : {str} Chaîne de caractères associée au *cpoly*. Cet argument est présent à des fins de rétro-compatibilité mais ne devrait pas être utilisé.
- **label** : {str} Chaîne de caractères descriptive qui sera affichée dans l'interface graphique.
- **min** : {int} Nombre de voix de polyphonie minimum.
- **max** : {int} Nombre de voix de polyphonie maximum.
- **init** : {int} Nombre de voix de polyphonie initial.

## 11.3.4 cgraph

Cette fonction crée une ligne de graphe qui représente l'évolution d'une variable sur la durée totale de la performance. Le graphique permet d'élaborer des automations complexes sur différents paramètres du processus sonores implémenté dans le module. On récupère la valeur continue de la ligne, en audio, à l'aide de la variable "self.name", où "name" doit être remplacé par la valeur donnée à l'argument *name* de la fonction *cgraph* concernée. Cette variable est créée automatiquement par la classe *BaseModule*. L'argument *func* permet de spécifier la forme initiale de la ligne en donnant une liste de points en format (temps - valeur). Les valeurs de temps doivent être spécifiées entre 0 et 1. Elles seront ensuite ajustées à la durée totale au lancement de la performance.

Si la valeur *True* est donnée à l'argument *table*, la fonction dessinée dans le graphique sera chargée en mémoire dans une table (objet *PyoTableObject*) plutôt que lue en continu. La variable "self.name" contiendra alors une table en mémoire et non une variable audio.

### Exemple

```
class Module(BaseModule):
    def __init__(self):
        BaseModule.__init__(self)
        self.wave = Osc(self.mytable, freq=self.freq, mul=.2)
        self.out = Mix(self.wave, voices=self.nchnls, mul=self.env)

Interface = [
    # Enveloppe d'amplitude (variable continue)
    cgraph(name="env", label="Overall Amp", func=[(0,1),(1,1)], col="blue1"),
    # Forme d'onde (en memoire dans une table)
    cgraph(name="mytable", label="Waveform Shape", min=-1, max=1, table=True,
           size=32768, func=[(0,0),(.25,1),(.75,-1),(1,0)], col="red1"),
    # Trajectoire de frequence (variable continue, echelle logarithmique)
    cgraph(name="freq", label="Base Frequency", min=20, max=10000, rel="log",
           func=[(0,250),(1,250)], col="orange1"),
]
```

**Déclaration**

**cgraph**(*name*="graph", *label*="Envelope", *min*=0.0, *max*=1.0, *rel*="lin", *table*=False, *size*=8192, *unit*="x", *curved*=False, *func*=[(0, 0.), (.01, 1), (.99, 1), (1, 0.)], *col*="red")

**Arguments**

- **name** : {str} Chaîne de caractères associée à la ligne de graphe. Une variable *self.name* sera créée par la classe *BaseModule*.
- **label** : {str} Chaîne de caractères descriptive qui sera affichée dans le menu déroulant du graphique.
- **min** : {float} Valeur minimum de la ligne sur l'axe Y.
- **max** : {float} Valeur maximum de la ligne sur l'axe Y.
- **rel** : {str} Type d'échelle utilisée pour la résolution de la ligne. "lin" = échelle linéaire, "log" = échelle logarithmique.
- **table** : {boolean} Si cet argument a la valeur *True*, une table sera créée plutôt que la lecture du ligne en continu. Permet de créer des enveloppes qui ne sont pas dépendantes de la durée de la performance (ex. enveloppe des grains dans un module de granulation).
- **size** : {int} Si l'argument *table* a la valeur *True*, *size* indique la taille de la table en échantillons.
- **unit** : {str} Description de l'unité de la ligne. N'est pas utilisée.
- **curved** : {boolean} Si cet argument a la valeur *True*, la ligne initiale sera dessinée à l'aide de segments courbes plutôt que linéaire. Même effet qu'un double-clic sur la ligne.
- **func** : {list of tuples} Liste de pairs de valeurs (X, Y) indiquant la forme initiale de la ligne. Les valeurs en X représente le temps, normalisé entre 0 et 1 tandis que les valeurs en Y représente l'amplitude des points, en fonction du minimum et du maximum.
- **col** : {str} Couleur associée à la ligne. Voir la liste des couleurs disponibles en fin de section.

### 11.3.5 cslider

La fonction *cslider* ajoute un potentiomètre au panneau situé sous le graphique de l'interface. Chaque potentiomètre se voit automatiquement attribué une ligne de graphe avec laquelle seront enregistrées/éditées/rejouées les automations pour le paramètre en question. On récupère la valeur courante du potentiomètre, en audio, à l'aide de la variable "self.name", où "name" doit être remplacé par la valeur donnée à l'argument *name* de la fonction *cslider* concernée. Cette variable est créée automatiquement par la classe *BaseModule*. L'argument *func* permet de spécifier une automation initiale au potentiomètre en donnant une liste de points en format (temps - valeur). Les valeurs de temps doivent être spécifiées entre 0 et 1. Elles seront ensuite ajustées à la durée totale au lancement de la performance.

Si la valeur *True* est donnée à l'argument *up*, aucune variable continue ne sera créée et une valeur discrète sera donnée à un appel de méthode à chaque relâche de la souris (après un changement de valeur sur le potentiomètre). La méthode doit être définie dans la classe "Module" et avoir le nom "self.name_up", où "name" doit être remplacé par la valeur donnée à l'argument *name* de la fonction *cslider* concernée.

Pour la définition de potentiomètre suivante :

```
cslider(name="num", label="# of Grains", min=1, max=256, init=32, gliss=0,
        res="int", up=True, unit="grs")
```

On devrait retrouver, dans la classe "Module", une méthode telle que :

```
def num_up(self, value):
    gr = int(value)
    # do whatever you want here...
```

À tout moment, on peut récupérer la valeur courante du potentiomètre, en data, à l'aide de la méthode *get()* de la variable créée pour le potentiomètre. Pour le *cslider* défini ci-dessus, on récupérerait le nombre de grains comme ceci :

```
nGrains = int(self.num.get())
```

### Déclaration

cslider(*name*="slider", *label*="Pitch", *min*=20.0, *max*=20000.0, *init*=1000.0, *rel*="lin", *res*="float", *gliss*=0.025, *unit*="x", *up*=False, *func*=None, *midictl*=None, *half*=False, *col*="red")

### Arguments

- **name** : {str} Chaîne de caractères associée au potentiomètre. Une variable *self.name* sera créée par la classe *BaseModule*.
- **label** : {str} Chaîne de caractères descriptive qui sera affichée à gauche du potentiomètre.
- **min** : {float} Valeur minimum du potentiomètre.
- **max** : {float} Valeur maximum du potentiomètre.
- **init** : {float} Valeur initiale du potentiomètre.
- **rel** : {str} Type d'échelle utilisée pour la résolution du potentiomètre. "lin" = échelle linéaire, "log" = échelle logarithmique.
- **res** : {str} Résolution numérique du potentiomètre. "float" = valeurs décimales, "int" = valeurs entières uniquement.
- **gliss** : {float} Durée du portamento audio, en secondes, appliquée aux valeurs produites le potentiomètre.
- **unit** : {str} Description de l'unité du potentiomètre. S'affiche à droite de la valeur numérique.
- **up** : {boolean} Si cet argument a la valeur *True*, le potentiomètre fournira une nouvelle valeur seulement sur la relâche de la souris. Une méthode est alors appelée avec la valeur courante en argument.
- **func** : {list of tuples} Liste de pairs de valeurs (X, Y) indiquant la forme initiale de l'automation associée dans le graphique. Les valeurs en X représente le temps, normalisé entre 0 et 1 tandis que les valeurs en Y représente l'amplitude des points, en fonction du minimum et du maximum. Si une liste est donnée à cet argument, le mode lecture du potentiomètre sera automatiquement activé.
- **midictl** : {int} Assigne, à l'initialisation, un contrôleur midi au potentiomètre.
- **half** : {boolean} Indique au potentiomètre d'utiliser la moitié de l'espace graphique d'un potentiomètre standard. Habituellement regroupé par deux.
- **col** : {str} Couleur associée au potentiomètre. Voir la liste des couleurs disponibles en fin de section.

### 11.3.6 crange

### 11.3.7 csplitter

### 11.3.8 ctoggle

### 11.3.9 cpopup

La fonction *cpopup* génère un menu déroulant, inséré dans le panneau à gauche des potentiomètres, permettant la sélection d'items dans une liste prédéfinie.

Deux variables associées au menu seront automatiquement créées par la classe *BaseModule*. Les noms de variable seront construits comme suit :

- **self.name + "_index"** : Un entier qui indique la position de l'item sélectionné dans le menu.
- **self.name + "_value"** : La chaîne de caractères de l'item choisi.

Où "name" doit être remplacé par la valeur donnée à l'argument *name* de la fonction *cpopup* concernée. Si la valeur "foo" est donné à *name*, les variables seront nommées *self.foo_index* et *self.foo_value*.

Si la valeur "k" est donnée à l'argument *rate*, une méthode portant le nom du menu (argument *name*) doit être définie dans la classe. La méthode, qui sera appelée automatiquement par la sélection d'un nouvel item dans le menu, recevra deux valeurs en argument, l'index et la chaîne de caractères sélectionnée. Exemple de déclaration de la méthode :

```
def foo(self, index, value):
    # index -> int
    # value -> str
```

#### Exemple

```
class Module(BaseModule):
    def __init__(self):
        BaseModule.__init__(self)
        self.noise = Noise(.1)
        self.filt = Biquad(self.noise, freq=1000, q=2, type=self.type_index)
        self.out = Mix(self.filt, voices=self.nchnls, mul=self.env)

    def type(self, index, value):
        self.filt.type = index

Interface = [
    cgraph(name="env", label="Overall Amp", func=[(0,1),(1,1)], col="blue1"),
    cpopup(name="type", label="Filter Type", init="Bandpass", col="green1",
           value=["Lowpass", "Highpass", "Bandpass", "Bandstop"], rate="k")
]
```

**Déclaration**

**cpopup**(*name*="popup", *label*="Chooser", *value*=["1", "2", "3", "4"], *init*="1", *rate*="k", *col*="red")

**Arguments**

- **name** : {str} Chaîne de caractères associée au menu déroulant. Une méthode portant ce nom doit être créée dans la classe audio.
- **label** : {str} Chaîne de caractères descriptive qui sera affichée à gauche du menu.
- **value** : {list of str} Liste de chaînes de caractères spécifiant les valeurs à insérer dans le menu déroulant.
- **init** : {str} Chaînes de caractères indiquant la valeur initiale du menu.
- **rate** : {str} Indique si le menu est dynamique ("k") ou actif seulement au lancement de la performance ("i").
- **col** : {str} Couleur associée au menu déroulant. Voir la liste des couleurs disponibles en fin de section.

### 11.3.10 cbutton

### 11.3.11 cgen

### 11.3.12 Liste des couleurs

Cinq couleurs, en quatre tons, sont disponibles pour le design de l'interface. On assigne la couleur désirée, en format chaîne de caractères, à l'argument *col* des fonctions de création d'éléments d'interface graphique. Voici la liste des couleurs possibles :

| | | | | |
|---|---|---|---|---|
| **red1** | **blue1** | **green1** | **purple1** | **orange1** |
| **red2** | **blue2** | **green2** | **purple2** | **orange2** |
| **red3** | **blue3** | **green3** | **purple3** | **orange3** |
| **red4** | **blue4** | **green4** | **purple4** | **orange4** |

## 11.4 Presets

## 11.5 Exemples

### 11.5.1 Filtre à état variable

Le premier exemple illustre l'utilisation d'un *sampler/looper* comme source audio d'un traitement. Le *sampler* implémente une lecture en boucle d'un fichier son avec contrôle sur les paramètres de la lecture. Le signal est ensuite passé au travers d'un filtre à état variable contrôlé par les potentiomètres de l'interface.

```
1 # Nul besoin d'importer pyo puisque le fichier sera execute a l'interieur de
2 # Cecilia, dans un environnement ou pyo est deja importe.
3
4 # Definition de la classe audio. Le nom *doit absolument* etre "Module" et doit
```

```
# aussi heriter de la classe "BaseModule" (definie a l'interne de Cecilia).
# C'est la classe "BaseModule" qui se charge d'initialiser les variables
# necessaires a la communication entre l'interface et la classe audio.
class Module(BaseModule):
    # Le string de documentation de la classe peut etre consulte dans Cecilia
    # avec la commande Ctrl+I (Cmd+I sous OSX)
    """
    State Variable Filter.

    This module implements lowpass, bandpass and highpass filters in parallel
    and allow the user to interpolate on an axis lp -> bp -> hp.

    Sliders under the graph:

        - Cutoff/Center Freq : Cutoff frequ for lp and hp (center freq for bp)
        - Filter Q : Q factor (inverse of bandwidth) of the filter
        - Type (lp->bp->hp) : Interpolating factor between filters
        - Dry / Wet : Mix between the original signal and the filtered signal

    Dropdown menus and toggles on the bottom left:

        - # of Voices : Number of voices played simultaneously (polyphony),
                        only available at initialization time
        - Polyphony Chords : Pitch interval between voices (chords),
                             only available at initialization time

    Graph only parameters :

        - Overall Amplitude : The amplitude curve applied on the total duration
                              of the performance
    """
    # Methode "constructeur", possede seulement "self" comme argument
    def __init__(self):
        # Appel de la methode "constructeur" de la classe parente
        BaseModule.__init__(self)

        # La methode "addSampler" de la classe BaseModule permet de recuperer
        # le signal provenant d'un sampler (lecture en boucle avec crossfade)
        # defini dans l'interface. Le string donne a la methode fait reference
        # au nom du "csampler", defini dans la liste de controles graphiques.
        self.snd = self.addSampler("snd")

        # Filtre a etat variable (interpole entre lp, bp et hp). Le filtre est
        # applique sur le signal du sampler (self.snd) et utilise 3 variables
        # crees automatiquement par la classe BaseModule: self.freq, self.q et
        # self.type. Ces variables ont ete crees en fonction de l'argument
        # "name" des trois sliders definis dans l'interface.
        self.dsp = SVF(self.snd, self.freq, self.q, self.type)

        # "self.out" *doit absolument* etre le nom de variable du signal audio
        # de la fin de la chaine de traitement. Cecilia recupere cette variable
        # pour faire suivre le signal audio vers la section Post-Processing et
        # ultimement, a la sortie audio. "self.drywet" est la variable associe
        # au cslider(name="drywet", ...) et "self.env" est la variable associee
```

```
        # au cgraph(name="env", ...) definis dans l'interface.
        self.out = Interp(self.snd, self.dsp, self.drywet, mul=self.env)

# Definition de l'interface graphique.
# Le nom de la liste *doit absolument* etre "Interface"
Interface = [
    # csampler genere un lecteur en boucle de fichiers sons
    csampler(name="snd"),
    # cgraph cree une ligne dans le grapher
    cgraph(name="env", label="Overall Amp", func=[(0,1),(1,1)], col="blue1"),
    # cslider genere un potentiometre avec sa ligne de graph associee
    cslider(name="freq", label="Cutoff/Center Freq", min=20, max=20000,
            init=1000, rel="log", unit="Hz", col="green1"),
    cslider(name="q", label="Filter Q", min=0.5, max=25, init=1, rel="log",
            unit="x", col="green2"),
    cslider(name="type", label="Type (lp->bp->hp)", min=0, max=1, init=0.5,
            rel="lin", unit="x", col="green3"),
    cslider(name="drywet", label="Dry / Wet", min=0, max=1, init=1, rel="lin",
            unit="x", col="blue1"),
    # cpoly met en place les deux menus servant a la gestion de la polyphonie
    cpoly()
]

# Tout ce qui suit le marqueur suivant dans un fichier .c5 a ete
# ecrit par l'application et concerne la sauvegarde des presets.

####################################
##### Cecilia reserved section #####
#### Presets saved from the app ####
####################################
```

scripts/chapitre_11/StateVarFilter.c5

### 11.5.2 Boucleur de son avec *feedback*

Le second exemple illustre l'utilisation d'une table permettant d'accomplir des processus sur un son chargé en mémoire vive. Un oscillateur dont le signal de sortie module la position de son propre pointeur de lecture est utilisé pour créer une modulation de fréquence d'un son par lui-même. Le signal passe ensuite par filtre afin d'adoucir le résultat sonore. Notez l'utilisation du menu déroulant (*cpopup*) pour sélectionner le type de filtre désiré.

```
class Module(BaseModule):
    """
    Self-modulated frequency sound looper.

    Sliders under the graph:

        - Transposition : Transposition, in cents, of the input sound
        - Feedback : Amount of self-modulation in sound playback
        - Filter Frequency : Frequency, in Hertz, of the filter
        - Filter Q : Q of the filter (inverse of the bandwidth)

    Dropdown menus and toggles on the bottom left:
```

```
13
14          - Filter Type : Type of the filter
15          - # of Voices : Number of voices played simultaneously (polyphony),
16                          only available at initialization time
17          - Polyphony Chords : Pitch interval between voices (chords),
18                               only available at initialization time
19
20      Graph only parameters :
21
22          - Overall Amplitude : The amplitude curve applied on the total duration
23                                of the performance
24
25      """
26      def __init__(self):
27          BaseModule.__init__(self)
28
29          # Creation de la table (SndTable) a partir du son selectionne.
30          # La methode addFilein se charge d'assurer que le nombre de canaux de
31          # la table est consequent avec le nombre de canaux de Cecilia.
32          self.snd = self.addFilein("snd")
33
34          # Conversion des valeurs donnees en "cents" en facteur de transposition
35          # (CentsToTranspo). On multiplie ici par "self.polyphony_spread" qui
36          # est la liste de transposition donnee par le menu "cpoly", en fonction
37          # de l'accord choisi et du nombre de voix de polyphonie. "self.trfactor"
38          # contiendra donc autant de streams audio qu'il y a de valeurs dans la
39          # liste "self.polyphony_spread".
40          self.trfactor = CentsToTranspo(self.transpo, mul=self.polyphony_spread)
41
42          # Les facteurs de transpo sont multiplies par la frequence a laquelle
43          # la table doit etre lue pour obtenir la hauteur originale du son.
44          self.freq = Sig(self.trfactor, mul=self.snd.getRate())
45
46          # OscLoop fonctionne comme SineLoop, mais on lui donne la table a lire
47          # en argument (le fichier son ici). Le slider de feedback est multiplie
48          # par une petite valeur sinon le resultat est trop chaotique.
49          # "self.polyphony_scaling" est une valeur d'amplitude qui s'ajuste pour
50          # diminuer le volume quand le nombre de voix augmente (pour garder une
51          # amplitude constante).
52          self.dsp = OscLoop(self.snd, self.freq, self.feed*0.0002,
53                             mul=self.polyphony_scaling * 0.5)
54
55          # On mix le signaux selon nombre de canaux de Cecilia (self.nchnls).
56          self.mix = self.dsp.mix(self.nchnls)
57
58          # Variable de sortie (self.out). Un filtre avec controle en slider
59          # (self.filt_f, self.filt_q). "self.filt_t_index" est une variable cree
60          # par la classe BaseModule qui contient la position de l'item choisi
61          # dans le menu "filt_t". "self.env" est la ligne de graph pour
62          # l'amplitude globale.
63          self.out = Biquad(self.mix, freq=self.filt_f, q=self.filt_q,
64                            type=self.filt_t_index, mul=self.env)
65
66      # Le menu deroulant appelle a chaque manipulation une methode qui porte
```

```
67      # son nom ("filt_t" dans ce cas-ci). On doit absolument definir cette
68      # methode, avec les arguments index et value, pour que le module fonctionne.
69      def filt_t(self, index, value):
70          """
71          index -> position de l'item dans le menu (entier)
72          value -> valeur selectionnee dans le menu (string)
73          """
74          # On assigne l'index a l'argument "type" du Biquad
75          self.out.type = index
76
77  Interface = [
78      cfilein(name="snd"),
79      cgraph(name="env", label="Overall Am", func=[(0,1),(1,1)], col="blue1"),
80      cslider(name="transpo", label="Transposition", min=-4800, max=4800, init=0,
81              unit="cnts", col="red1"),
82      cslider(name="feed", label="Feedback", min=0, max=1, init=0.25, unit="x",
83              col="purple1"),
84      cslider(name="filt_f", label="Filter Frequency", min=20, max=18000,
85              init=10000, rel="log", unit="Hz", col="green1"),
86      cslider(name="filt_q", label="Filter Q", min=0.5, max=25, init=1, rel="log",
87              unit="x", col="green2"),
88      # cpopup genere un menu deroulant dans le panneau en bas a gauche
89      cpopup(name="filt_t", label="Filter Type", init="Lowpass", col="green1",
90             value=["Lowpass", "Highpass", "Bandpass", "Bandreject"]),
91      cpoly()
92  ]
```

scripts/chapitre_11/FeedLooper.c5

# Chapitre 12

# Retour sur les principaux concepts

## 12.1 Éléments de langage de la programmation orientée objet en Python

— Syntaxe des noms de variables
— Typage des variables
— Affectation de valeurs
— Opérateurs de comparaison
— Séquence d'instructions et exécution conditionnelle
— Instructions imbriquées
— Règles de syntaxe
— Les chaînes de caractères (*string*)
— Les opérations sur les listes
— La fonction *range*
— Importation de modules
— Instructions répétitives
— Les générateurs de listes
— Les fonctions
— Variables locales et variables globales
— Les dictionnaires
— Les classes

## 12.2 Principes généraux de la programmation musicale avec le module *pyo*

— Premier pas

— Server

— PyoObject, manuel

— PyoTableObject

— Comment lire le manuel *pyo*

— Les paramètres d'initialisation

— SfPlayer et les *paths*

— Gestion de la polyphonie

— Utilisation des listes à des fins musicales

— variations continues des paramètres

— Utilisation des fonctions dans un script (object *Pattern*)

— Génération d'une séquence composée (object *Score*)

— Gestion du temps par l'envoi de *triggers*

— Exemples musicaux

— Exemple de classe audio

# Chapitre 13

# Annexes

## 13.1 Interaction avec l'utilisateur

Lorsque l'on démarre l'environnement Python, une banque de fonctions intégrées sont accessibles immédiatement. Ce sont les fonctions 'built-in', dont font partie **range**, **len** et **type** que nous avons déjà rencontrées. Il existe, dans ce module, deux fonctions permettant à l'utilisateur d'interagir avec le programme en cours d'exécution. Ce sont les fonctions **input** et **raw_input**.

### 13.1.1 La fonction *input*

Lorsque Python rencontre cette fonction, il arrête l'exécution du programme et attend que l'utilisateur inscrive une valeur et tape la touche ≪Return≫. La valeur est mémorisée dans une variable et le type le plus approprié lui est assigné. Un argument, sous la forme d'une chaîne de caractères, doit être donné à la fonction **input** afin de guider la réponse de l'utilisateur. Elle sera affichée à l'écran juste avant le curseur en attente.

```
>>> prenom = input('Entrez votre prenom : ')
Entrez votre prenom : 'Olivier' # (on inscrit une valeur puis <Return>)
>>> print prenom
Olivier
>>> type(prenom)
<type 'str'>
>>> val = input('Entrez un nombre : ')
Entrez un nombre : 1000
>>> print val
1000
>>> type(val)
<type 'int'>
```

Notez l'utilisation des guillemets à la ligne 'Entrez votre prenom'. Sans l'usage des guillemets, l'interpréteur cherchera une variable *Olivier* et s'il n'en trouve pas, il rapportera une erreur.

### 13.1.2 La fonction *raw_input*

La fonction **raw_input** offre plus de flexibilité pour l'utilisateur. Avec cette fonction, nul besoin de se soucier du type de la réponse donnée à la question, la fonction convertit automatiquement en *string* la valeur obtenue.

```
>>> prenom = raw_input('Entrez votre prenom : ')
Entrez votre prenom : Olivier
>>> print prenom
Olivier
>>> type(prenom)
<type 'str'>
>>> val = raw_input('Entrez un nombre : ')
Entrez un nombre : 1000
>>> print val
1000
>>> type(val)
<type 'str'>
```

On trouvera les fonctions **int** et **float** parmi les fonctions intégrées. Elles servent respectivement à convertir un *string* en nombre entier ou en nombre réel.

```
>>> val = raw_input('Entrez un nombre : ')
Entrez un nombre : 1000
>>> type(val)
<type 'str'>
>>> val = int(val)
>>> type(val)
<type 'int'>
```

## 13.2 Gestion des exceptions

Lorsqu'une erreur survient dans le déroulement des instructions, une exception est signalée. Le programme arrête et un message d'erreur est affiché. Il est parfois essentiel d'effectuer des opérations qui peuvent, sous certaines circonstances, s'avérer impossibles à exécuter. Le système de gestion des exceptions sous Python est simple et permet facilement d'attraper les exceptions avant qu'elles n'arrêtent l'exécution du programme. Un autre chemin peut alors être spécifié pour les cas où une opération aurait échoué. Voici un exemple où le programme arrête lorsque l'on essaie d'ouvrir un fichier inexistant :

```
>>> f = open('new.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'new.txt'
```

### 13.2.1 Interception d'une exception (permet au programme de continuer)

En introduisant notre opération dans un bloc d'instructions **try** - **except**, le programme tentera d'exécuter la commande dans le bloc **try** et, en cas d'échec, exécutera plutôt les commandes inscrites dans le bloc **except**. Dans l'exemple suivant, le programme tente d'ouvrir le fichier 'new.txt'. S'il réussit, il saute le bloc **except** et poursuit le déroulement normal du programme. S'il échoue, il commence par afficher le message de l'erreur survenue et il ouvre ensuite un fichier qui se nomme 'default.txt', en le plaçant dans la variable où devait se trouver le fichier 'new.txt'. Le programme peut continuer à rouler avec les informations se trouvant dans 'default.txt', prévu pour les cas de catastrophe.

```
try:
    f = open('new.txt', 'r')
except IOError, msg:
    print msg
    f = open(default.txt, 'r')
```

La constante 'IOError' placée après le mot clé **except** indique au bloc de n'intercepter que ce type d'erreur. Dans un premier temps, il est plus prudent de ne rien indiquer après le mot clé, le bloc interceptera alors toutes les exceptions possibles.

```
try:
    f = open('new.txt', 'r')
except:
    f = open(default.txt, 'r')
```

Une clause **else** peut être rajoutée à la suite de la clause **except**. Cette clause ne sera exécutée que si le **try** fonctionne.

Si la clause **finally** est ajoutée à la toute fin du bloc, elle sera exécutée peut importe le résultat de l'opération précédente.

```
try:
    f = open('new.txt', 'r')
except:
    f = open(default.txt, 'r')
else:
    print 'Le fichier new.txt existe bien'
finally:
    print 'Fin du bloc'
```

## 13.3 Envois de commandes au système

Il existe plusieurs techniques pour envoyer des commandes au système, certaines étant dépendantes de la plate-forme sur laquelle le script Python est exécuté. Nous ne ferons ici qu'un survol d'une technique simple, fonctionnant sur toutes les plate-formes, c'est à dire la fonction *system* du module *os*.

La commande *os.system* fonctionne très bien pour des opérations simples ne nécessitant aucun retour dans le programme Python. La commande suivante permet de démarrer le logiciel QuickTime à partir d'un interpréteur Python :

```
>>> import os
>>> os.system('open /Applications/QuickTime\ Player.app')
```

La commande suivante donnera la liste des fichiers présents dans le répertoire courant, avec les informations détaillées, et sauvera le tout dans le fichier 'repertoire.txt' :

```
>>> os.system('ls -al > repertoire.txt')
```

Les module *os* et *os.path* regorgent de fonctions permettant de gérer les répertoires sur le disque. Des opérations telles que renommer ou créer un dossier, construire des liens (paths) vers des fichiers ou changer les permissions doivent, dans la mesure du possible, être effectuées avec les fonctions génériques de Python, afin d'assurer le caractère portable du programme. En effet, la syntaxe n'étant pas la même sur les différents systèmes, il est préférable de laisser cette charge à Python, qui s'assurera d'être conforme à la plate-forme sur laquelle il est lancé. Ainsi, pour ouvrir un fichier, la fonction *os.path.join* permet de construire le lien vers un fichier sans gérer la syntaxe :

```
>>> import os.path
>>> f = open(os.path.join('Resources', 'default.py'), 'r')
>>> f.read()
'setGlobalDuration(12)\n\nplaySound()\n\nstartCsound()\n'
```

On peut donner autant de niveaux que nécessaires à la fonction *os.path.join*, en format *string*, séparés par des virgules.

Pour lancer des opérations systèmes plus complexes, il faudra utiliser les fonctions du module **subprocess**, qui remplace tous les appels système depuis Python 2.5.

## 13.4 L'antisèche Python (*Python Cheat Sheet*)

### 13.4.1 Arithmétique

#### Types

Integer : 12, -234, 0, ...
Long integer : 54323457L (on ajoute le suffixe 'L')
Float : 0.005, 13.4654, -4.321, ...

#### Opérateurs

Exposant : ** ( 2**4 = 16 )
Multiplication : *
Division : / ( 5/2 = 2, 5/2. = 2.5 )
Addition : +
soustraction : -
Modulo : %
Ordre de priorité : PEMDAS
(parenthèse, exposant, multiplication, division, addition, soustraction)

#### Expressions abrégées

$a \mathrel{+}= 1 \iff a = a + 1$
$a \mathrel{*}= 5 \iff a = a * 5$

### 13.4.2 Assignation

Le symbole '=' permet d'assigner une valeur à une variable.
La relation d'égalité est indiquée par le symbole '=='. (voir Conditions 13.4.9)

#### Assignation simple

a = 25 (assigne la valeur 25 à la variable *a*)

#### Assignations multiples

a = b = c = 1 (assigne la valeur 1 aux variables *a*, *b* et *c*)

#### Assignations parallèles

a, b = 1, 2 (assigne la valeur 1 à la variable *a* et la valeur 2 à la variable *b*)

### 13.4.3 Chaîne de caractères (string)

Un *string* est une constante ; pour modifier un *string* il faut en créer un autre. Il peut être délimité par des guillemets simples (') ou des guillemets doubles ("). Les guillemets triples ("' ou """) servent à définir un *string* de documentation.

#### Opérations

Assignation (=) : str = 'Ceci est un string'
Concaténation (+) : 'Hello ' + 'world !' $\Longrightarrow$ 'Hello world !'
Répétition (*) : 3 * 'allo' $\Longrightarrow$ 'alloalloallo'

Les guillemets simples et doubles sont équivalents, mais peuvent servir à introduire un *string* dans un *string* :

```
>>> print 'Ceci est un "string" dans un string'
Ceci est un "string" dans un string
```

#### Construction d'un string avec l'opérateur modulo

```
>>> a, b, c = 1, 3, 7
>>> print '%d, %d et %d sont des nombres premiers' % (a, b, c)
1, 3 et 7 sont des nombres premiers
```

#### Quelques opérations sur les strings

```
>>> a = 'allo'
>>> a.capitalize()
ALLO
>>> " ".join(a)
A L L O
>>> a.replace('L', 'B')
ABBO
>>> a.replace('O', 'A')
ABBA
>>> ' '.join(a).split()
['A', 'B', 'B', 'A']
>>> a[0]
'A'
>>>a[1:3]
'BB'
```

### 13.4.4 Lecture et écriture de fichiers

#### Ouvrir un fichier en lecture

```
f = open('Users/olipet/mon_fichier.txt', 'r')
for line in f.readlines():
    print line
f.close()
```

**Ouvrir un fichier en écriture**

```
f = open('Users/olipet/mon_fichier.txt', 'w')
f.write('Hello world!')
f.close()
```

### 13.4.5 Print

La commande print permet d'afficher des valeurs dans le *stdout* ou dans l'interpréteur.

```
>>> a = 25
>>> print a
25
>>> print 'Hello world!'
Hello world!
>>> print "L'addition de 2 + 2 donne: %d" % (2+2)
L'addition de 2 + 2 donne 4
```

Inclure les lignes suivantes dans votre script pour faire suivre le *stdout* vers un fichier de votre choix.

```
import sys
sys.stdout = open("outputlog.txt", "w")
```

### 13.4.6 Liste

Une liste consiste simplement en une suite de valeurs séparées par des virgules et placées entre crochets []. Les éléments dans une liste peuvent être de différents types (nombre, string, liste, tuple, dictionnaire, fonction, objet, ...). La liste est altérable, c'est à dire que l'on peut modifier ses éléments sans avoir à créer une nouvelle liste. Voici quelques opérations sur les listes :

- list1 + list2 : concaténation de list1 et list2
- list[i] : retourne l'élément à la position i
- list[i :j] : retourne une liste contenant les éléments entre les positions i et j
- len(list) : retourne la longueur de la liste
- del list[i] : élimine l'élément à la position i
- list.append(val) : ajoute l'élément *val* à la fin de la liste
- list.extend(list) : ajoute une liste à la fin d'une liste
- list.sort() : met dans l'ordre les éléments d'une liste
- list.reverse() : inverse les éléments d'une liste
- list.insert(i, val) : insère *val* à la position i

— list.count(val) : retourne le nombre d'occurrences de *val* dans une liste

— list.pop() : retourne et élimine la dernière valeur d'une liste

— val in list : True si l'élément *val* est présent dans la liste

### 13.4.7 Tuple

Un tuple, tout comme la liste, est une suite de valeurs séparées par des virgules. Par contre, un tuple est délimité par des parenthèses et est immuable. Particulièrement utile pour créer des listes qui ne doivent en aucun cas être modifiées, le tuple est aussi plus rapide d'accès que la liste.

### 13.4.8 Dictionnaire

Le dictionnaire est une table de données sur base de pairs clé - valeur. Un dictionnaire est défini entre accolades {}. Les pairs sont séparées par des virgules et la clé est séparée de sa valeur par le symbole deux-points ( :).

```
mon_dict = {'un': 123,
            12: [1,2,3,4,5,6],
            (1,2): 'Hello World!'}

>>> print mon_dict['un']
123
>>> print mon_dict[12]
[1,2,3,4,5,6]
>>> print mon_dict[(1,2)]
Hello world!
```

Quelques opérations sur un dictionnaire :

```
>>> dict = {'Montreal': 514, 'Quebec': 418}
>>> dict['Sherbrooke'] = 819
>>> print dict['Sherbrooke']
819
>>> del dict['Quebec']
>>> dict
{'Montreal': 514, 'Sherbrooke': 819}
>>> dict.keys()
['Montreal', 'Sherbrooke']
>>> dict.has_key('Montreal')
True
>>> 'Sherbrooke' in dict
True
```

Pour des clés qui sont de simples *strings*, il est possible de spécifier les pairs clé-valeur en donnant des mots-clés à la fonction "constructeur" de la classe dictionnaire.

```
>>> mon_dict = dict(Montreal=514, Quebec=418, Sherbrooke=819)
>>> print mon_dict
{'Montreal': 514, 'Quebec': 418, 'Sherbrooke': 819}
```

Les éléments ne sont pas classés en ordre dans un dictionnaire, ce qui signifie qu'il est impossible de prédire l'ordre d'arrivé des éléments dans une boucle **for**.

### 13.4.9 Conditions (if... elif... else)

#### Opérateurs de comparaison

< : plus petit que
<= : plus petit ou égal à
> : plus grand que
>= : plus grand ou égal à
== : est égal à
! = : n'est pas égal à

#### Syntaxe

Les blocs d'opérations sont délimités par l'indentation du programme :

```
if a < 0:
    print 'a est plus petit que 0, a prend la valeur 0'
    a = 0
elif a > 100:
    print 'a est plus grand que 100, a prend la valeur 100'
    a = 100
else:
    print 'a est compris entre 0 et 100'
```

Utilisation de **and**, **or** et **not** :

```
if a >= 0 and a <= 100:
    print 'a est compris entre 0 et 100 inclusivement'

if a < 0 or a > 100:
    print 'a est soit plus petit que 0, soit plus grand que 100'

if not a == 0:
    print 'a ne vaut pas 0'

if a != 0:
    print 'a ne vaut pas 0'

if a not in [1,2,3,4,5,6,7,8,9,10]:
    print 'a n'est pas une valeur entre 1 et 10'
```

### 13.4.10 Boucle (for, while)

La boucle **for** sert à 'incrémenter' dans les membres d'une séquence (liste, tuple, dictionnaire).

```
>>> for i in range(5):
...     print i
...
0
1
2
3
4
>>> for name in ['Paul', 'Jack', 'Joe']:
...     print name
...
Paul
Jack
Joe
```

Boucle **for** sur les clés d'un dictionnaire :

```
>>> dict = {'jaune': 'citron', 'rouge': 'fraise', 'vert': 'lime'}
>>> for key in dict.keys():
...     print key, dict[key]
...
jaune citron
rouge fraise
vert lime
```

La boucle **while** sert à construire une boucle de longueur indéterminée.

```
>>> var = 1
>>> while var > 0:
...     var = random.randint(0,100)
```

### 13.4.11 'List comprehension' (fabrication de listes)

La fabrication de listes est une des fonctionnalité les plus puissantes du langage Python. Elle permet de créer des listes complexes très rapidement par une utilisation particulière de la boucle **for**. Les 'list comprehension' sont délimitées par des crochets.

#### Générateur simple

```
>>> ma_list = [i for i in range(20)]
>>> ma_list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

#### Générateur avec boucles for imbriquées

```
>>> ma_list = [(i,j) for i in range(3) for j in range(3)]
>>> ma_list
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

#### Générateur avec conditions

```
>>> ma_list = [(i,j) for i in range(3) for j in range(3) if j != 1]
>>> ma_list
[(0, 0), (0, 2), (1, 0), (1, 2), (2, 0), (2, 2)]
```

### 13.4.12 Création de fonctions

Une fonction est définie par le mot-clé **def**. Toutes les lignes suivantes qui sont au niveau d'indentation supérieur feront partie de la fonction.

#### Fonction simple

```
>>> def ma_fonction():
...     print 'fonction simple'
...
>>> ma_fonction()
fonction simple
>>> def carre(x):
...     print x*x
...
>>> carre(2)
4
```

#### Fonction avec arguments possédant une valeur par défaut

Les arguments qui ont une valeur par défaut doivent toujours être après les arguments sans valeur par défaut.

```
>>> def puissance(x, exp=2):
...     print x**exp
...
>>> puissance(2)
4
>>> puissance(2, 3)
8
>>> puissance(x=3, exp=4)
81
```

#### Fonction avec valeur de retour

Une fonction avec le mot-clé **return** permet de récupérer une valeur à la sortie d'une fonction.

```
>>> def puissance(x, exp=2):
...     return x**exp
...
>>> a = puissance(2, 3)
>>> a
8
```

#### *String* de documentation

Un *string* entre triple guillemets à la ligne suivant la déclaration de la fonction sera enregistré dans la variable __doc__ de la fonction.

```
>>> def puissance(x, exp=2):
...     '''Calcule et retourne la valeur de x a la puissance exp'''
...     return x**exp
...
>>> puissance.__doc__
'Calcule et retourne la valeur de x a la puissance exp'
```

### 13.4.13 Création de classes

Une classe est une collection de méthodes référant à l'objet défini par la variable 'self' en premier argument. Une classe permet de créer plusieurs objets similaires mais complètement indépendants les uns des autres.

#### La méthode 'constructeur'

C'est la méthode qui sera exécutée à la création de l'objet ( __init__ ).

```
class Cercle:
    def __init__(self, r):
        self.rayon = r
```

**Les autres méthodes** de l'objet ont accès aux variables commençant par l'objet lui-même (self).

```
import math

class Cercle:
    def __init__(self, r):
        self.rayon = r

    def circonference(self):
        return math.pi * (2 * self.rayon)

    def aire(self):
        return math.pi * self.rayon**2
```

Pour utiliser les méthodes d'une classe, il suffit de créer un objet et d'appeler ses méthodes avec la syntaxe *objet.méthode()*. Par exemple, considérant que la classe précédente a été sauvegardée dans le fichier Cercle.py :

```
>>> from Cercle import Cercle
>>> a = Cercle(4)
>>> a.circonference()
25.132741228718345
>>> a.aire()
50.26548245743669
```

### 13.4.14 Module

Un module est une collections de fonctions réunies dans un fichier .py.
Pour utiliser ces fonctions il faut d'abord importer le module.

#### Première forme : importer un module

```
import sys, time
print sys.path
print time.time()
```

#### Deuxième forme : importer des éléments d'un module

```
from random import randint, uniform
print randint(100,200)
print uniform(0,1)
```

#### Importer toutes les fonctions d'un module

```
from random import *
```

#### Modifier le nom de référence

```
import wx.html as html

win = html.HtmlWindow()
```

### Créer ses propres modules

Dans le fichier monmodule.py :

```
"""Mon module de fonctions"""
def un():
    print 'fonction un'

def deux(x):
    print 'le carre de %d est: %d' % (x, x*x)
```

Dans un interpréteur :

```
>>> import monmodule
>>> monmodule.un()
fonction un
>>> monmodule.deux(2)
le carre de 2 est: 4
>>> dir(monmodule)
['__builtins__', '__doc__', '__file__', '__name__', 'un', 'deux']
>>> monmodule.__doc__
'Mon module de fonctions'
```

La fonction *dir()* renvoi la liste des attributs et des fonctions appartenant au module.

### 13.4.15 Exceptions

Lorsqu'une erreur survient dans le déroulement des instructions, une exception est signalée. Le programme arrête et un message d'erreur est affiché.

```
>>> f = open('new.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'new.txt'
```

**Interception d'une exception (permet au programme de continuer à rouler)**

```
try:
    f = open('new.txt', 'r')
except IOError, msg:
    print msg
    f = open(default.txt, 'r')
```

### 13.4.16 Mots clés

| | |
|---|---|
| **and** | "et", permet de combiner des conditions |
| **assert** | outil de "déboguage", affiche les erreurs de déclaration |
| **break** | permet de sortir d'une boucle avant la fin |
| **class** | démarre la définition d'une classe |
| **continue** | avance directement à la prochaine itération d'une boucle |
| **def** | démarre la définition d'une fonction ou d'une méthode |
| **del** | élimine un objet et sa référence |
| **if** | démarre une condition |
| **elif** | rajoute une possibilité dans une déclaration conditionnelle |
| **else** | exécution si aucune condition n'a été respectée |
| **try** | tentative d'exécuter des instructions |
| **except** | si *try* n'a pas réussi, permet d'intercepter des exceptions |
| **finally** | exécution à la fin d'un bloc *try... except... finally* (optionnel) |
| **exec** | exécution dynamique de code Python |
| **for** | déclare une boucle |
| **from** | désigne un module duquel importer des classes ou des fonctions |
| **global** | déclare une variable globale |
| **import** | importe des modules, des classes ou des fonctions |
| **in** | permet de tester l'appartenance d'une variable à un groupe |
| **is** | permet de tester l'identité d'une variable |
| **lambda** | création d'une fonction anonyme |
| **not** | négation, permet des conditions négatives, ex. if a not in list |
| **or** | "ou", permet de combiner des conditions |
| **pass** | "passe tout droit", aucune exécution |
| **print** | affiche des valeurs |
| **raise** | affiche des messages d'erreur ou d'exception |
| **return** | met fin à une fonction et, si nécessaire, retourne une valeur |
| **while** | déclare une boucle à durée indéterminée |
| **yield** | gèle l'état d'un générateur jusqu'au prochain appel 'next' |
| **None** | valeur nulle |
| **True** | booléen 'vrai' |
| **False** | booléen 'faux' |