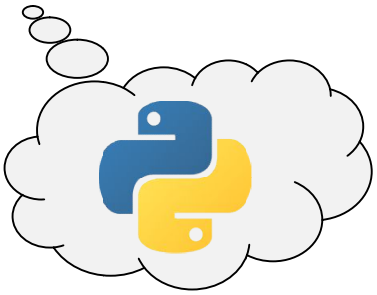
FOXDOT >> JOUER DE LA MUSIQUE GRÂCE A PYTHON



Marvin WORTMAN

B3 SDS
CampusID

Contribution Période 2
7 Janvier 2019

2013876@campusid.eu
wortman.marvin@gmail.com

Table des matières

I. Fonctionnement de ce document

Ce document est destiné à être lu en version numérique. Il comprend à plusieurs reprises des lignes de code qui peuvent être simplement copiées-collées dans l'IDE approprié. Les lignes de code sont écrites en une police différente, les commentaires portent une couleur violette.

Plusieurs paragraphes feront référence à une source ou à d'autres chapitres du document. Pour naviguer d'une référence à l'autre, vous pouvez utiliser l'outil de recherche (Ctrl+F), chaque item référencé portera le même nom en majuscules entre crochets; par exemple ce paragraphe contient la référence unique [DOC_INFO].

Pour ajouter un peu plus de lien entre le public "programmeur" et le public "musicien", un glossaire et quelques annexes sont à votre disposition.

Ce document est disponible publiquement sur les Internets, et un article de blog sur le site http://idboard.net:10000/web/2019/01/06/le-live-coding-la-performance-artistique-par-le-code en fait un résumé.

Toute distribution payante de ce document, ne sera certainement pas au goût du créateur de FoxDot et de sa documentation.

Le contenu présenté fait seulement part des connaissances actuelles de l'auteur. Pour toute réclamation ou correction sur ce contenu, veuillez envoyer un e-mail à l'adresse indiquée en page de garde.

II. Histoire de FoxDot : l'aventure du Live Coding

L'histoire de FoxDot commence avant sa création, avec l'avènement du "Live Coding". Le Live Coding désigne une pratique transformant le fait d'écrire du code, en performance, exécutée en direct devant une audience, à des fins artistiques ou bien éducatives.

Le Live Coding plébiscite une énorme communauté internationale de concepteurs et d'artistes en tous genres, au Brésil, au Canada, au Royaume-Uni, aux USA, en Allemagne. Des performances sont aussi organisées sur les plateformes Internet consacrées, comme Twitch ou Youtube Live. Une scène musicale est apparue sous le nom d'Algorave, et réunit les sympathisants lors de concerts.

De nombreuses applications existent dans le but de produire de la musique en codant.
Un des langages les plus évolués pour ce faire est SuperCollider. SuperCollider est un moteur musical extrêmement complet disposant de son propre langage de programmation. Il intègre des fonctions de synthèse complexes, qu'il faut prendre le temps de bien maîtriser avant de pouvoir espérer composer un morceau. En réponse à cela, plusieurs applications de Live Coding sont apparues, qui sont des sous-ensembles, des modules, permettant de communiquer avec SuperCollider.

Ces applications permettent une expérience utilisateur moins rébarbative, et une courbe d'apprentissage plus rapide. A l'état actuel du marché, plusieurs applications existent : **TidalCycles** (Haskell), **Sonic Pi** (Ruby), **ixi lang** (sclang), **ChucK**, et notamment **FoxDot**.

FoxDot a été créée en 2015 par Ryan Kirkbride. Ce créateur est doctorant en musique à l'Université de Leeds, et a commencé à la conceptualiser après des tentatives jugées insatisfaisantes sur TidalCycles.

En effet, FoxDot se démarque de ses congénères par le fait qu'il base son fonctionnement sur l'utilisation de python. Python est un langage très répandu, et a la réputation d'être simple à enseigner. FoxDot offre un appel de synthétiseurs et de percussions élégant et simple à écrire.

III. Principe de base de FoxDot

FoxDot est une interface de programmation en python, permettant de simplifier le processus d'écriture fastidieux proposé par SuperCollider. Pour se lancer il faut donc disposer des deux environnements : SuperCollider et python. FoxDot est disponible comme package python, pour l'installer il suffit d'utiliser le *pip*. A son exécution, FoxDot propose son propre IDE. Chaque commande peut être écrite et exécutée avec le raccourci Ctrl+Enter, et un terminal intégré écrit les retours de chacune des instructions.

FoxDot et SuperCollider doivent fonctionner en même temps car chaque commande écrite sous FoxDot va être interprétée et transmise à SuperCollider !

Note : FoxDot est aussi disponible comme package à part entière sur le store de certains IDE, comme Atom.

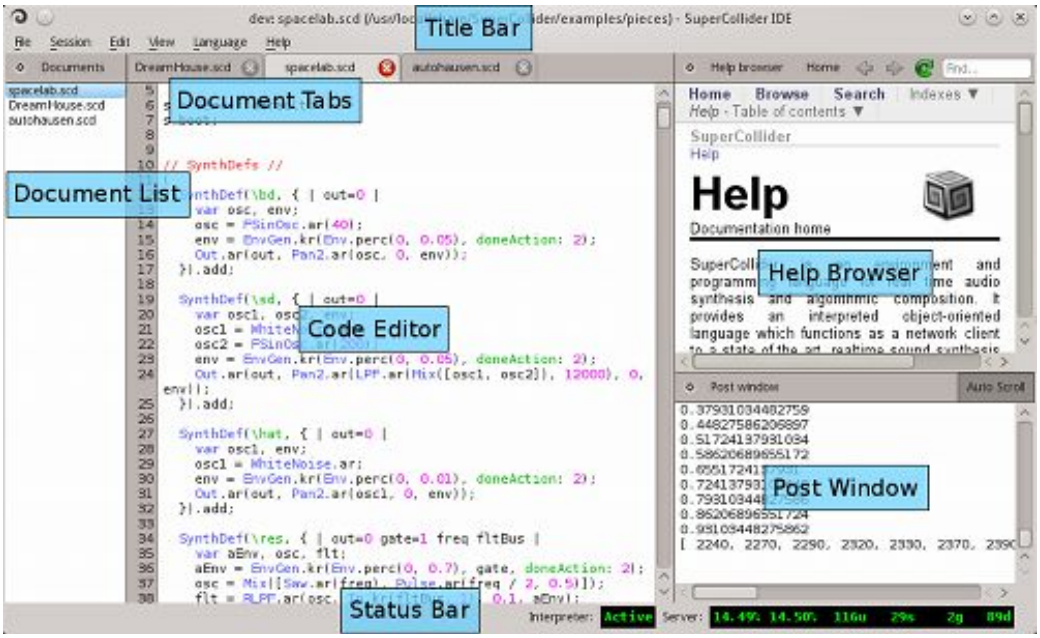
IV. SuperCollider, qu'est-ce ?

Quasiment aucune instruction n'est écrite sur SuperCollider par les utilisateurs lors de leur expérience avec FoxDot, néanmoins il est intéressant d'en proposer quelques bases, afin d'apprécier la différence entre les deux syntaxes.

D'après le github officiel du projet, SuperCollider est une plateforme de composition algorithmique, utilisée par les artistes et les chercheurs en son. C'est un serveur audio en temps réel, contenant des outils pour l'analyse, la synthèse et la production; d'un langage interprété : le **sclang**; et de l'IDE où seront écrites les instructions; ainsi que de l'interpréteur.

[SOURCES]

Cette plateforme est installée sous Windows par le biais d'un exécutable, et s'ouvre comme tel. Pour installer SuperCollider et FoxDot, tout est détaillé dans la procédure d'installation.



On voit que l'interface de SuperCollider dispose d'un module où sont écrites les instructions, d'un module d'aide et d'un module de terminal, qui rend compte en temps réel des actions effectuées. Une barre de statut nous permet de visualiser l'état du serveur audio.

Attention :

Lors de l'exécution de code générant du son, que l'on se trouve sous SuperCollider ou FoxDot, **attention au volume sonore pratiqué** lors des expérimentations : une mauvaise surprise peut rapidement arriver, surtout quand on débute et qu'on ne sait pas quelles seront les caractéristiques des sons générés.

Ci-dessous un snippet de code extrait de la documentation. Nous allons nous en servir pour dégager quelques observations relatives à la syntaxe du sclang.

```
// 1
"Hello World!".postln;

// 2
(
"Live Coding music,".postln;
"it’s fantastic.".postln;
)

// 3
s.boot;
Server.local.boot;
s.quit;

// 4
f = { arg a; a.value + 3 };
f.value(3);
g = { 3.0.rand; };
f.value(g);
f.value(g);

// 5
(
f = { arg a, b;
    var firstResult, finalResult;
    firstResult = a + b;
    finalResult = firstResult * 2;
    finalResult;
};
f.value(2, 3);
)

// 6
{ SinOsc.ar(440, 0, 0.2) }.play;

// 7
(
{ var freq;
    freq = [[660, 880], [440, 660], 1320, 880].choose;
    SinOsc.ar(freq, 0, 0.2);
}.play;
)

// 8
{ Pan2.ar(PinkNoise.ar(0.2), SinOsc.kr(0.5)) }.play;

// 9

(
SynthDef(\smooth, { |out, freq = 440, sustain = 1, amp = 0.5|
    var sig;
    sig = SinOsc.ar(freq, 0, amp) * EnvGen.kr(Env.linen(0.05, sustain,
0.1), doneAction: Done.freeSelf);
```

```
    Out.ar(out, sig ! 2)
}).add;
)

(
p = Pbind(
    \instrument, \smooth,
    \midinote, Pseq([60, 72, 71, 67, 69, 71, 72, 60, 69, 67], 1),
    \dur, Pseq([2, 2, 1, 0.5, 0.5, 1, 1, 2, 2, 3], 1)
).play;
)
```

1 : L'affichage du fameux "Hello World!"; ici on voit que l'appel d'affichage de texte en console est "**postln**". On reconnaît aussi la syntaxe des commentaires à une ligne : "**//**", utilisée dans d'autres langages tels JavaScript ou PHP, ainsi que le point virgule qui termine chaque ligne.

2 : Nous sommes dans un environnement qui exécute le code en direct. Placer le curseur sur une ligne de code et appuyer Ctrl+Enter ou Shift+Enter, permet d'exécuter cette ligne de code uniquement. L'intérêt d'avoir des parenthèses est de sélectionner plusieurs lignes de code à exécuter séquentiellement. L'interpréteur les exécutera une à une, de haut en bas.

3 : Le serveur audio doit être démarré avant toute exécution de script musical, sinon aucun son ne sera produit. Les deux commandes de boot sont valables et provoquent le démarrage du serveur. La fonction .**quit** permet de stopper le serveur.

4 : Les fonctions sont écrites sous forme d'objets; le mot-clé "**arg**" définit les arguments; la méthode **.rand** génère un nombre aléatoire (dans ce cas, entre 0 et 3); la méthode **.value** est celle permettant d'appeler la fonction.

5 : Le mot-clef **var** permet de définir des variables locales dans la fonction. On remarque aussi qu'ici la valeur à retourner ne nécessite pas de mot-clef **return**.

6 : Nécessite que le **serveur soit actif**. La méthode **.play** permet de jouer de manière continue, une onde sinusoïdale (instance de **SinOsc**), de fréquence 440Hz, de phase initiale 0, de multiplicateur 0.2. Le multiplicateur peut être vu comme une amplitude de volume (hauteurs des pics). D'autres classes de générateurs sont disponibles : **VarSaw, SyncSaw, Blip, Saw, Pulse**,...

7 : L'écriture entre crochets représente un tableau ou array. La méthode **.choose** permet de choisir aléatoirement un des éléments du tableau. Cette méthode va donc jouer indéfiniment une suite de notes aléatoires, certaines sont des accords à deux tons.

8 : La classe **Pan2** permet d'appliquer de manière périodique un passage du canal stéréo gauche au canal stéréo droit.

9 : Exemple plus complexe. Ces instructions sont destinées à jouer la chanson "Over the Rainbow". Le **SynthDef** est utilisé pour créer un nouvel instrument virtuel en lui donnant certaines propriétés. **PBind** est le gestionnaire de mélodie; il joue la mélodie selon les règles données par ses arguments : nature des instruments à utiliser, suite de notes et valeurs rythmiques des notes.

V. Installer FoxDot

A la lumière de ces différents exemples, on peut remarquer que pour commencer à vraiment "faire de la musique", les pré-requis sont nombreux. On doit comprendre les différentes classes et outils proposés pour la création d'instruments (**SynthDefs**), trouver la sonorité désirée, et ensuite seulement passer à la conception des "partitions", l'écriture des suites de notes et marqueurs de rythme désirées.

FoxDot permet de commencer à expérimenter avec une gamme variée de **SynthDefs** déjà fournis. Avant de commencer à ordonner les différents **players** de FoxDot, il faut le "connecter" à SuperCollider.

[SOURCES]

En premier lieu, on suit la suite de la procédure d'installation, puis :
1/ Sur Terminal : `pip install FoxDot`
2/ Sur SuperCollider : `Quarks.install("FoxDot");`
3/ `FoxDot.start`
4/ Sur Terminal : `python -m FoxDot`

La commande `FoxDot.start` va aussi vérifier si le serveur est actif, et, si ce n'est pas le cas, exécuter un `s.boot`.

Le serveur audio est à présent à l'écoute, la communication entre les FoxDot et SuperCollider est établie, et nous pouvons donc commencer notre design de sons.

Une fois qu'un player joue en playback sous FoxDot, un objet Clock compte le nombre de **beats** (noires) et de **bars** (mesures) qui passe, chaque player joue ses instructions en boucle.

Recommandation d'usage :
Parfois, la composition sur FoxDot nécessite trop de ressources mémoire et résulte en un bip très pénible, qui ne peut être arrêté que par la fermeture de FoxDot ou l'extinction du playback. Pour prévenir l'incident, il faut changer l'allocation mémoire du serveur son en cours sur SuperCollider :
1/ `FoxDot.stop`
2/ `Server.local.options.memSize = 8192`
3/ `s.reboot`
4/ `FoxDot.start`

Si les incidents persistent, cette valeur doit être à nouveau augmentée.

Note aux musiciens :
Sur une **partition**, on doit écrire une **armure** et une **signature de temps** pour savoir dans quelle **tonalité** et à quel **rythme** le morceau sera joué. Le **tempo** doit être donné en indiquant la durée d'une **noire**. Sur un logiciel de composition (**DAW**), on peut régler le tempo, la signature de temps, et positionner les notes sur une interface visuelle **midi** ressemblant à un piano.

Sur FoxDot, chacun de ces paramètres doit être déclaré avant de commencer une session.
Sinon, FoxDot choisit ceux par défaut : un tempo de 120 bpm, la tonalité de Do Majeur, et une signature de temps 4/4.

FoxDot étant destiné à faire des sessions d'improvisation en utilisant des **motifs mélodiques** de petite taille,

il n’est pas forcément aisé de jouer un morceau en suivant une partition ou une tablature. Cela nécessite d’autant plus de planification lorsqu’on n’a pas de connaissance solide du solfège.

Nous allons voir comment s'initier pas à pas aux différents outils permettant de jouer une chanson.

[SOURCES]

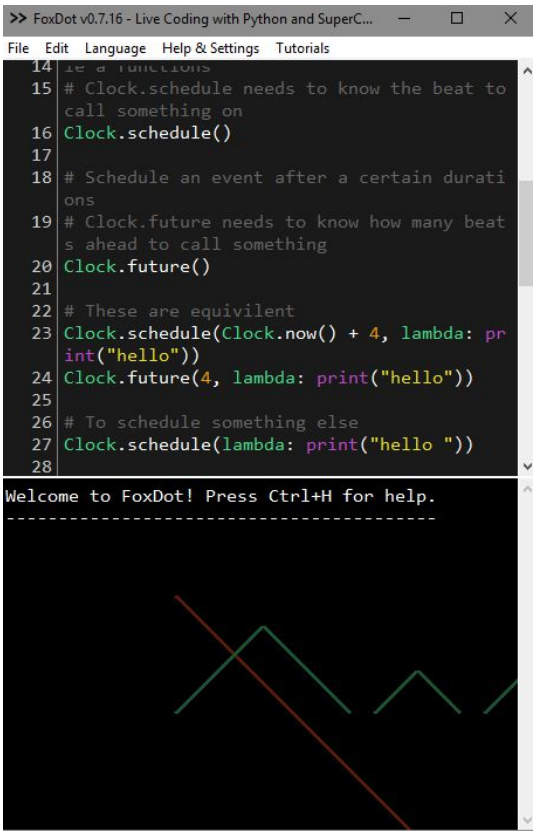
Pour cela, je choisis de livrer l’information dans un ordre un peu différent de celui de la documentation officielle. En effet, nous allons d’abord revoir les caractéristiques de base de notre session; pour dans un second temps voir plus en détails comment écrire des **motifs percussifs**, des **successions d'accords** et des **motifs mélodiques**.

Si FoxDot et SuperCollider sont majoritairement utilisés lors de performances live, la documentation SuperCollider permet aussi de trouver comment enregistrer une session dans un fichier **wav**. On peut donc imaginer d’autres usages intéressants comme extraire un pattern ou un son particulier, pour s’en servir ultérieurement comme **sample**.

VI. Découvrons l'API FoxDot

FoxDot peut être vu comme une API python, au même titre que celle utilisée pour manipuler les capteurs d'un senseHat. En effet les mots-clefs que nous allons utiliser dépendent de **classes** qui leur sont associées.
FoxDot, à l’instar de SuperCollider, est un **runtime**. Ce qui veut dire que chaque ligne qui est écrite, sera exécutée à l’appui de Alt+Enter. Pour exécuter plusieurs lignes à la fois, il suffit de les laisser groupées (pas de saut de ligne) et d'appuyer Ctrl+Enter.

L’interface graphique de FoxDot se décompose en deux parties : un IDE qui permet de garder chaque ligne écrite en mémoire; ainsi qu’une partie terminal, qui fournit les retours d’exécution :



L'instruction "Export console log" du menu Edit est très pratique pour enregistrer des informations relatives à notre session !

Ci-dessous, je vais proposer à nouveau moult des lignes de code avec leurs commentaires, n'hésitez pas à les copier-coller dans votre IDE FoxDot et à les exécuter pas-à-pas, pour voir comment il réagit !

[SOURCES]

FoxDot dispose par défaut d'une bibliothèque de guides, accessibles depuis le menu **Tutorials**. Ces guides détaillent par thématique les explications données ici-même, ainsi que dans la documentation officielle.

Pour sauter les parties théoriques, rendez-vous au chapitre suivant : Expérimentation musicale.

A/ Constantes de notre chanson

```
# Afficher le tempo
print(Clock.bpm)
# Changer le tempo à 110 pour la session en cours
Clock.bpm = 110

# Afficher toutes les tonalités disponibles dans l'API
print(Scale.names())
# Changer la tonalité de la session en cours en gamme mineure
Scale.default.set("minor")
# Identique à :
Scale.default = "minor"
```

[AN_TEMP][AN_TONALITE]

```
# FoxDot permet aussi de créer des tempéraments personnalisés
# La commande ci-dessous met en place une gamme majeure en tempérament
occidental standard (division de l'octave en 12) et est aussi le réglage
par défaut :
Scale.default.set([0, 2, 4, 5, 7, 9, 11], tuning=[0, 1, 2, 3, 4, 5, 6, 7,
8, 9, 10, 11, 12])

# On peut ajouter pour la session en cours des attributs à nos objets Scale
et Tuning, afin de les retrouver plus rapidement
Scale.custom = [0,1,4,5,6,8,10]
Tuning.custom = [0, 1, 2, 3, 4, 5, 6.5, 7, 8, 9, 10, 10.5, 12]

# On peut jouer avec la signature de temps :
Clock.time_signature = (4,4)

# Pour manipuler la note racine, on choisit un chiffre correspondant à la
liste de notes de notre tuning (par défaut c'est l'échelle chromatique) :
Root.defaut.set(2)
```

B/ Utilisation de samples et formation de motifs percussifs

[AN_RYTHM][AN_PAT]

```
# Bien, nous avons un tempo, un tempérament, une tonalité et une signature de temps, jouons quelques percussions à présent

# De quelles percussions disposons-nous ?
print(Samples)

# Nous sommes en phase d'expérimentations, donc nous allons souvent avoir besoin de jouer des morceaux, et de les arrêter
b1 >> play("x")
b1.stop()
Clock.clear()
```

La méthode .stop() arrête la lecture du player désigné. Les variables pré-enregistrées pour exécuter des players sont des combinaisons de chiffres et de lettres en minuscule. On tape le nom de notre variable suivi de deux chevrons droits. Attention, un nom de variable commençant par un chiffre ou constitué d'une seule lettre ou chiffre, n'est pas valable. La méthode Clock.clear() permet de stopper tous les players en cours. Son raccourci clavier est Ctrl+. (oui, le caractère : "point").

```
# On choisit une trame rock simple : des charlets, une grosse caisse et une caisse claire
d1 >> play("x o ")
d2 >> play("----")

# Cette forme est la plus simple pour combiner deux motifs percussifs en un seul, il en existe d'autres :
# L'utilisation de patterns, qui ont la forme P[] : nous y reviendrons plus tard
d0 >> play(P["x o "] & P["----"])

# L'utilisation de chevrons, comme suit :
d0 >> play("<x o ><->")

# Pour ne pas se décaler dans le temps au fil des loops, les contenus des chevrons doivent faire des tailles identiques ou proportionnelles (exemple : 4 et 8 pour un playback en 4/4)
# d1 peut être écrite en groupant mes items par parenthèses, ce qui offre une lecture par rubriques. Les espaces ( ) sont nécessaires car ils permettent de jouer un silence
d1 >> play("(x )(o )")

# Si je veux donner plus de pep's à d2, je peux lui permettre de jouer différents charlets dans le temps nécessaire à en jouer un seul, pour cela je les groupe avec des crochets :
d2 >> play("([--]---)")
# Tout sample situé entre crochets est compressé autant de fois que nécessaire pour rester dans le timing voulu;
# Cet effet de compression peut être très utile pour jouer des patterns cadencés de type breakcore ou drum'n'bass

# Enfin, une autre option permettant d'ajouter de la vie à un pattern, est de jouer aléatoirement un élément par rapport à un autre :
```

```
d0 >> play("<x o x o ><{-[--]}------->")
# Ici mon premier sample de charlet peut être à chaque nouvelle lecture du
pattern remplacé aléatoirement par un double-charlet


# Le mot-clef "play" correspond à un player. Un player est une classe
disposant de plusieurs méthodes et attributs;
# Qu'est-il possible de faire avec les méthodes et les attributs ?
# Les attributs sont donnés comme arguments à mon objet play, séparés par
des virgules, et les méthodes peuvent être appliquées de manière successive
à un même objet, en les séparant par des points
```

[AT_DUR][AN_RYTHM][AN_PAT]

L'attribut que nous allons voir: "dur", est éminemment important puisque c'est grâce à lui que nous déterminons la durée relative où sont jouées nos notes et nos percussions.
La durée 1 en 4/4 est simplement une noire (ou un beat). Ce qui veut dire qu'en effet, une durée de 4 correspondra à une ronde. Le réglage par défaut des "dur" lorsqu'on joue un pattern via le mot-clef "play", est à ½. Cela représente une croche. La durée ¼ est une double-croche.
La durée de playback est directement liée à l'objet Clock. En effet; la durée de chaque note est détectée, et jouée au tempo demandé.

```
# Pour imager, le pattern suivant est identique à notre pattern d0 :
d1 >> play("xo", dur=1) # on a enlevé les espaces car ils sont contenus
dans la durée
d2 >> play("-") # la durée par défaut est toujours 1/2
```

Le player a une autre particularité intéressante, c'est qu'on peut lui ajouter de nouveaux sons au format .wav et en répertorier les caractères correspondants dans le fichier consacré : plus d'infos sur le tutorial 11 : Playing Custom Samples.

L'attribut "sample" est réservé au player de samples. Il s'agit d'un nombre, correspondant au numéro du dossier contenant les différents fichiers wav dans la hiérarchie. En effet, les partenaires de FoxDot ont fourni une banque de sons énorme, permettant une variété, des belles percussions réalistes aux sons de jeux vidéo rétro. Exemple ci-dessous :

```
# N'importe quel nombre peut-être mis, chaque nombre supérieur au nombre
réel de dossiers remettre le compteur à zéro
d3 >> play("<@ x ><:-:->", sample=8)

# Nous reviendrons ultérieurement aux attributs
```

C/ Les listes et les patterns dans FoxDot

[PATTERN][AN_PAT]

Avant de nous lancer dans la création des mélodies et des accords, revenons sur un outil très pratique proposé par FoxDot pour automatiser plusieurs tâches.

```
# FoxDot utilise les listes de python :
a = [1,2,3,4,5,6]

# Cependant un autre système a dû être mis en place à cause de la méthode
de calcul par défaut lorsque l'on ajoute deux listes ensemble :
a = [1,2,3,4,5,6]
b = [7,8,9]
print(b*2) # retourne [7,8,9,7,8,9]
print(a+b) # retourne [1,2,3,4,5,7,8,9]
# En effet il s'opère un ajout des éléments dans la liste, mais aucun
opération sur chacun des éléments n'est possible. Il faudrait donc un
algorithme capable d'itérer les opérations voulues sur chaque élément

# …Et cet algorithme est déjà en place, il se nome Pattern et s'écrit P[] :
c = P[1,2,3]
d = P[4,5,6]
print(c+d) # Retourne [1+4,2+5,3+6]
print(c-d) # Retourne [1-4,2-5,3-6]
# Et cela est possible avec chacun des opérateurs : *, /, %, +, -, &, |,...

# Mais que se passe-t-il lorsqu'on met en relation deux listes de longueurs
différentes ?
# L'itérateur Pattern parcourt à nouveau le début de la liste la plus
courte afin de terminer toutes les opérations demandées à la liste la plus
longue : le résultat est un pattern ayant le PPCM des deux patterns
initiaux comme longueur
e = P[4,7,8,9,5,6]
f = P[3,3,4]
print(e+f) # Retourne [4+3,7+3,8+4,9+3,5+3,6+4] : 6 éléments

# On peut aussi imbriquer plusieurs patterns l'un dans l'autre, les listes
imbriquées sont automatiquement interprétées reconnues comme patterns :
g = P[1,2,[3,4],5] # identique à [1,2,3,5,1,2,4,5]
h = P[1,[3,4],[5,6,7]] # identique à [1,3,5,1,4,6,1,3,7]

# On peut aussi fabriquer des listes et des patterns à partir d'une syntaxe
python spécifique aux listes d'entiers :
print(P[:8]) # Retourne P[0,1,2,3,4,5,6,7] (de 0 à 8-1)
print(P[2:4]) # Identique à P[2,3] (de 2 à 4-1)
print(P[2:15:3]) # Retourne P[2,5,8,11,14] (de 2 à 15-1 avec un pas de 3)
```

Les listes et les Patterns peuvent être utilisés indépendamment. Il faut choisir l'un ou l'autre en fonction des besoins. Les Patterns répondent bien sûr à plus de besoins puisqu'ils disposent de différentes méthodes utiles, en plus du calcul matriciel que nous venons de voir. Ces patterns peuvent être utilisés comme argument à tout moment dans la conception d'un morceau avec nos SynthDefs. Plus d'informations dans le tutorial 03 : Using Patterns.

Il existe aussi des fonctions intégrées à FoxDot, qui permettent de calculer des patterns automatiquement à partir d'algorithmes. Le Tous les algorithmes ne sont pas détaillés ici, mais un guide s'en charge au tutorial 15 : Pattern Generator Reference.

D/ Fabriquer des suites d'accords et des motifs mélodiques

Maintenant que nous savons utiliser nos patterns et que nous nous attendons à appliquer des arguments et des méthodes à nos players et à nos SynthDefs, nous sommes prêts à découvrir comment générer des notes de musique !

1. Manipulation basique des SynthDefs

Le player, qui permet de jouer les motifs percussifs, possède des fonctionnalités étendues lorsqu'on lui assigne un SynthDef. Le SynthDef est, comme nous en avons parlé, une définition d'instrument, fabriquée à l'aide de SuperCollider.

```
# Montrer tous les SynthDefs existants
Print(SynthDefs)
# Pour créer votre propre SynthDef, rendez-vous sur le tutorial 12 :
SynthDefs

# Nous allons choisir le SynthDef karp pour nos premiers exemples
# Par défaut tous les effets sont désactivés, la note jouée est 0, l'octave
est 5, la tonalité est celle définie par défaut, la durée de la note est 1,
le volume est à 100% (1), et c'est joué en boucle
ex >> karp()

# est identique à :
ex >> karp(degree=0, dur=1, oct=5, amp=1, bpm=Clock.bpm,
scale=Scale.default, tuning=Tuning.ET12, root=0, amplify=1, hpf=0, lpf=0,
bitcrush=0, drive=0, coarse=0, chop=0, cut=0, delay=0, formant=0, slide=0)

# Ci-dessus, j'ai dévoilé quelques caractéristiques et effets, qui peuvent
être appliqués si on leur fournit une valeur différente de 0. Il en existe
d'autres, la liste en est obtenue par les commandes :
print(Player.get_attributes())
print(FxList)
# Pour plus de détails sur leurs fonctionnements, rendez-vous dans le
tutorial 14 : Player Attributes Reference... N'hésitez pas non plus à les
expérimenter !

# Le degree d'un SynthDef peut être sous-entendu à condition qu'il soit le
premier argument :
ex >> karp(0, dur=1)

# identique à :
ex >> karp(dur=1, degree=0)


# Je peux modifier n'importe quel attribut de mon player actif durant le
playback, par exemple :
ex.degree=P[1,2,3]
```

```
# Je peux jouer uniquement ce player, pendant que les autres continuent de
jouer silencieusement
ex.solo()

# Pour désactiver le solo et reprendre le playback des autres players :
ex.solo(0)

# Pour arrêter les autres players complètement (une commande .solo(0) sera
sans effet) :
ex.only()

# Il est possible de lire l'état actuel d'un player actif par la méthode
.now :
print(ex.now(), ex.now("dur"))

# Il faut savoir qu'un attribut reste actif dans le playback même lorsque
je l'efface et que je ré-invoque mon player :
ex >> karp(0, dur=1, drive=0.05)

Je ne peux pas juste effacer la commande et ré-invoquer le même player :
ex >> karp(0, dur=1)

# Pour enlever l'overdrive je dois explicitement le remettre à  0 :
ex.drive = 0

# ou bien réinitialiser mon player :
ex.reset()

# Pour stopper ce player :
ex.stop()

# Pour couper tous les playbacks en cours :
Clock.clear()
```

2. Les accords

[AT_DUR] [PATTERN][AN_PAT]

Jouer des accords fait appel à un autre objet python bien connu : les tuples. Similaires à des listes, il s'agit d'une succession d'éléments enfermés entre parenthèses. Les tuples ont aussi une classe pattern associée, plus d'informations dans le tutorial 4 : Using Patterns.

```
# Pour plus d'aide sur les Patterns, on peut invoquer la commande :
help(Pattern)

# En général pour obtenir de l'aide sur n'importe quel objet, on appelle :
# Help(Object)
# Help(Object.method)

# Il y a plusieurs manières d'exécuter des accords en jouant les notes en
même temps, mais toutes font appel à des tuples :

# Mettre chaque tuple dans un pattern
ac >> space(P[(0,3,5), (1,4,6), (2,5,7), (4,7,9)], dur=4)

# Mettre chaque élément de chaque tuple dans un pattern différent
ac >> space(P(P[0,1,2,4], P[3,4,5,7], P[5,6,7,9]), dur=4)

# Ajouter le tuple à chaque note du pattern, depuis l'extérieur du player;
cette méthode est la moins verbeuse si on souhaite jouer exactement le même
accord à chaque fois
ac >> space(P[0,1,2,4], dur=4) + (0,3,5)
# Le choix d'une durée de 4 permet d'avoir un long accord, soit une ronde
en 4/4
```

3. Les mélodies

Une mélodie est jouée en faisant appel à des listes ou bien à des patterns. Différentes méthodes et attributs permettent d'intervenir sur les notes que nous souhaitons jouer. [AN_PAT]

```
# Reprenons notre suite d'accords en tonalité de Do# mineure :
Scale.default.set("minor")
Root.default.set(1)
ac >> space(P[0,1,2,4], dur=4, root=1) + (0,3,5)

# On lui ajoute un motif percussif, en prenant garde d'avoir un nombre
d'éléments compatible
ry >> play("x-{ox}-{hv} *-",dur=1, sample=6, echo=0.2,room=2, hpf=500,
hpr=0.85)
# Ici, on a bien 8 éléments qui ont une durée de 1, un motif percussif
couvre 8 noires.

# Une méthode très pratique est follow. Elle évalue la note jouée
couramment par sa cible et la reproduit :
m1 >> blip(dur=1, amp=2, oct=7).follow(ac) + (0,-3,-5)

# Ici, ac est en train de jouer un accord. Pour en obtenir une seule note,
on doit donc annuler le tuple en retranchant ses valeurs. On obtient un
tuple possédant les mêmes hauteurs de notes :
print(m1.degree) # Retourne un triplet (n,n,n)

# Cette mélodie est un peu plate, non ? Ajoutons-y un pattern
m1 >> blip(dur=1, amp=2, oct=7).follow(ac) + (0,-3,-5) + P[1,2,8,2,0,1]
# Le pattern ajouté compte 6 membres. On obtient un nouveau pattern
comptant le ppcm de 4 et 6, à savoir 12. 12 notes d'une durée de 1, cela
couvre donc 12 noires. Ma succession d'accords compte 16 noires, on a donc
induit un petit décalage entre l'accord et la mélodie, qui ne s'annulera
que lorsqu'on atteindra un nombre de mesure multiple de 12 et de 16.

# On peut enlever le décalage en ayant le bon nombre de membres au pattern
à ajouter:
m1 >> blip(dur=1, amp=2, oct=7).follow(ac) + (0,-3,-5) + P[1,2,8,2,0,1,0,2]

# On peut aussi définir un pattern complètement différent, en ne suivant
pas la note de mes accords :
m1 >> blip(P[P[1,2],P[3,8],1], dur=[1,1/2], amp=2, oct=7)
```

[AT_DUR][PATTERN][AN_PAT]

Il peut rapidement devenir ardu de compter exactement le nombre de notes jouées durant l'exécution de la suite d'accords. On a établi qu'il y a 16 noires de couvertes sur ces accord (4 accords de durée 4 chacun);

On a écrit un motif mélodique pouvant se résumer à : P[1,3,1,2,8,1], ce qui compte 6 notes. Ces 6 notes sont jouées en boucle en attribuant ½ comme durée à chaque seconde note, on se retrouve alors avec :

1 + ½ + 1 + ½ + ...

```
# Le but est de déterminer si notre pattern va se décaler ou si la fin d'un des loops correspond bien à la fin de la suite d'accords, alors comptons grâce à python :
print((1 + 1/2)*11) # Retourne 16.5

# 16.5 pour des itérations de durée à 1.5, cela signifie qu'il n'y a aucun moyen d'atteindre 16 avec ces durées-là, on doit donc retravailler notre mélodie :
print((1 + 1/2 + 1/2)*8) # Retourne bien 16

# 8 est un multiple de 4, mais pas de 6, il faut donc retravailler à nouveau notre mélodie en enlevant des éléments du pattern des notes, et le résultat est :
m1 >> blip(P[P[1,2],P[3,8]], dur=[1,1/2, 1/2], amp=2, oct=7)
# A présent on est bien sûr de n'avoir aucun de décalage dans le playback, notre pattern va être répété exactement 4 fois durant notre progression d'accords

# Un autre outil aurait permis de déterminer résultat similaire :
m1 >> blip(P[P[1,2],P[3,8]], dur=PDur(3,8), amp=2, oct=7)

# Le Pattern PDur permet d'entrer en argument le nombre d'itérations de durée, ainsi que le nombre de notes total à jouer lors d'un loop. Ici 8 est un diviseur de 16 et un multiple de 4, donc c'est une valeur correcte, et 3 le nombre de changements de rythme que nous désirons au sein de notre pattern. Cet algorithme pondère les différentes durées
print(PDur(3,8)) # Retourne P[0.75,0.75,0.5]
# L'addition : 0.75 + 0.75 + 0.5 donne aussi 2, comme pour l'addition de nos durées dans le cas précédent

# Autres méthodes utiles :

# La méthode .spread consiste à changer la panoramique (par défaut à 0) pour donner un effet stéréo à notre player :
m1.spread()

# Pour la désactiver :
m1.spread(0)

# La méthode penta permet de régler l'échelle associée à notre player à sa forme pentatonique :
m1.penta()

# Pour revenir à l'échelle standard :
m1.penta(0)

# La méthode stutter permet de répéter la note sur laquelle son exécution se produit
m1.stutter() # la note à l'exécution de la méthode est répétée une fois

# On peut lui assigner aussi différents paramètres :
m1.stutter(8, dur=2, pan=[-1,1]) # la note à l'exécution est répétée 8 fois avec une durée 2 et un effet de panning
```

```
# Les méthodes modifiant le degree

m1.reverse() # permet d'inverser l'ordre de la liste de notes en cours (celle calculée après application du pattern)

m1.rotate() # tire le 2è élément de la liste au début tout en conservant l'ordre de la boucle (donc le 1er élément est mis à la fin)

m1.rotate(5) # la valeur par défaut est 1, mais on peut aussi l'utiliser sur un autre élément de la liste

m1.shuffle() # retourne au hasard un nouvel ordre pour les éléments de la liste

# Les méthodes de temporalité :
m1.every(3,"stutter") # permet d'invoquer un évènement tous les n beats

#every peut aussi prendre une liste ou un pattern comme paramètre :
m1.every([4,6],"reverse")

m1.sometimes("stutter") # appelle un évènement au hasard au cours du temps avec une faible probabilité

m1.often("stutter") # appelle un évènement au hasard  au cours du temps avec une forte probabilité

m1.never("stutter") # enlève l'évènement de la planification
```

E/ Faire n'importe quoi

Les moyens de "casser les règles" sont nombreux, en effet, pour s'éloigner d'une construction musicale conventionnelle, il suffit de varier différents paramètres au travers des différentes pistes, ce qui, usuellement, a la réputation de casser la cohérence du morceau, en plus d'être un sacré brouhaha.

Sur FoxDot cette liberté est là, on peut faire n'importe quoi, comme dans l'exemple ci-dessous.
A ce moment-là, il est nécessaire de faire preuve d'un bon esprit d'abstraction et d'un fin sens de l'improvisation pour parvenir à réellement suivre l'édifice que l'on essaie de construire, aussi expérimental soit-il. Par ailleurs, il serait beaucoup plus pénible de retranscrire une telle complexité sur une partition.

```
# La Clock est toujours réglée en 4/4. Imposer des durées fantaisistes rend le tout indigeste, tant le motif manque de régularité : chaque mesure contient un bloc différent. Il est aussi techniquement possible de jouer des notes ne correspondant pas à notre systèmetonal, en utilisant des nombres décimaux :
p1 >> play("la musique", dur=P[1/2,1/4])
p2 >> charm(P[0,1,2.5], dur=[1/3,1/3,0.8333335454,18/27], scale=Scale.yu)
p3 >> blip(P[2,3,4], dur=[1/2,87/88,101/87,77/45], scale=Scale.chromatic, root=6) + (0,3)
```

F/ Aller plus loin

Le contenu de ce document est bien insuffisant pour décrire l'intégralité des possibilités offertes par FoxDot. En plus des potentielles fonctionnalités que son créateur ajoutera à l'avenir, voici les différents concepts qui n'ont pas été abordés ici :

- Importation de samples
- Création et importation de SynthDefs
- Création et utilisation de fonctions personnalisées
- Planification de l'exécution de script en fonction du temps de playback
- Utilisation de variables qui changent de valeur en fonction du temps
- Approfondissement des algorithmes de création de patterns
- Approfondissement des méthodes et effets pouvant être appliqués aux players (percussifs et non-percussifs)
- Approfondissement des méthodes pouvant être appliquées aux patterns
- Envoi des messages sur l'état du playback à un dispositif midi
- Enregistrement des sessions en wav
- Utiliser un serveur distant pour des sessions collaboratives
- Exprimer un tempérament avec des valeurs en Hz, ou lieu de valeurs en cents

Les fonctionnalités offertes par FoxDot vont au-delà de la composition musicale, puisque grâce aux différents outils d'algorithmie dont nous disposons, il est rendu possible de créer des chansons aux rendus aléatoires en quelques lignes à peine, qui peuvent s'auto-piloter sur une durée indéterminée.

VII. Expérimentation musicale

Après la démonstration théorique, voici venu le temps d'appliquer tout ce que nous avons vu. Ci-dessous, le code non-commenté d'une session; tout est fait dans l'ordre, de haut en bas :

```
Clock.bpm = 120
Scale.default.set("minor")
Root.default.set(5)

u1 >> play("hihi",sample=var([4,7],3),dur=P[1/2,1,1/2],room=0.3)

po >> klank(P[1,5,3],dur=2,formant=var([0.02,0.05],8))

lu >> karp(dur=PDur(5,8), pan=PSine()).follow(po)

lu.every(4,"stutter",4,lpf=1000,oct=7, scale=Scale.major, root=6)

pi >> sawbass(var([0,-2,-4,1],8),dur=1,formant=0.05)

lk >> dirt(dur=PDur(5,8),oct=5, drive=0.05,lpf=1000).follow(lu)

lu.never("stutter")

Clock.bpm = linvar([120,140],8)

Scale.default.set("locrian")

lu.amplify=0

oi >> orient(P[0,5,6,8,7],dur=4,amp=[0,1],oct=4) + P(0,3,5)

ol >> play("#",dur=8, sample=4,room=0.5)

oi.stop()

li >> karp(dur=PDur(2,3)).follow(lk)

Scale.default.set("mixolydian")

ol.stop()

li.solo()

li.dur=PDur(1,2)

Clock.bpm=90
```

Il peut être utile de changer de tempo ou de tonalité au cours même de la chanson, afin d'ajouter un peu d'effets et de surprise. De la même manière, il ne faut pas hésiter à expérimenter les différents outils qu'offre FoxDot, comme le panning, les filtres passe-haut ou passe-bas, etc… Ce résultat ne prend pas en compte tous les ajustements ayant dû être faits. En bref, un seul mot : expérimenter et trouver ce qui vous plaît !

Conclusion

Ce guide pas-à-pas nous a permis de maîtriser les fonctionnalités de base de FoxDot pour en tirer un morceau de musique simple.

Le meilleur apprentissage pour maîtriser FoxDot ne réside pas que dans la documentation, il faut aussi avoir la curiosité de regarder et d'analyser les nombreuses sessions disponibles sur youtube. On devient meilleur au fur et à mesure de pratique et d'expérimentation, que cela soit au niveau de la composition en elle-même, avec le choix des tonalités et des mélodies, mais aussi au niveau des effets et des motifs percussifs, que l'on peut personnaliser à l'infini.

Je remercie M. Ryan Kirkbride ainsi que ses collaborateurs pour l'énorme effort déployé ces dernières années en conception, documentation et démonstration de ce projet, et espère qu'il fédèrera plus d'utilisateurs à travers le monde.

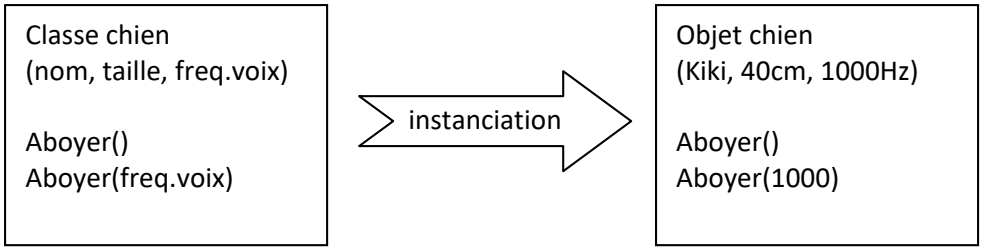
Annexe 1 : Glossaire

Player : Classe générale contenant les définitions des objets pouvant émettre des sons sous FoxDot

Classe : Ensemble d'attributs et de méthodes génériques

Objet : "Instance" de classe, où les attributs prennent corps.

Exemple :



Chaque instance de la classe "chien" va se voir assigner ses caractéistiques propres. De la même manière, lorsque j'instancie un objet player, je lui assigne un SynthDef, une suite de notes, des durées, etc…

Motif percussif: Suite répétitive et cohérente de percussions

Motif mélodique : Suite répétitive et cohérente de notes

Succession d'accords : Suite répétitive et cohérente d'accords (accord de base : trois notes)

Signature de temps : Ce qui permet de distinguer combien d'unités (noires) constituent une mesure (ronde)

Tuning : Système de référence couvrant les relations entre chaque note

PPCM : Plus Petit Commun Dénominateur, se calcule entre deux nombres; exemple pour 4 et 6 le PPCM est 12 : 12=2x6 et 12=3x4

Tempo : Nombre de beats par minute

Annexe 2 : Notions de théorie musicale

A/ Tempérament ou tuning

[AN_TEMP]

Un tempérament est l'espace dans lequel sont définies nos notes ou *pitchs*.
Celui le plus usité la division en douze notes de l'octave, qui constituent la gamme chromatique tempérée :
Do, Do#, Ré, Ré#, Mi, Fa, Fa#, Sol, Sol#, La, La#, Si

L'octave a en effet la propriété d'avoir la fréquence double de la note de l'octave inférieur, ce qui amène une similitude entre les sons perçus.

Dans les programmes générant des tempéraments, comme Scala, l'unité à utiliser est le cent. Le cent se définit comme le centième du demi-ton tempéré. Il y a douze demi-tons, et un octave vaut 1200 cents.
La formule mathématique du cent est donc la suivante :

$$c = 1 = \ 12 \ \times log_2\left(\frac{f_2}{f_1}\right)$$

Au niveau des fréquences, la référence est le La 3, qui est à 440 Hz.
Le rapport d'une fréquence à l'octave est de deux. Par exemple, pour passer du La 3, au La 4, on multiplie la fréquence de référence par deux, ce qui donne 880 Hz.

On peut déduire le rapport définissant un demi-ton (rapport entre deux fréquences successives) à partir de l'expression du cent :

$$\frac{f_2}{f_1} = \ r = \ 2^{\frac{100\times c}{12\times 100}} = 2^{\frac{c}{12}} = \sqrt[c\times 12]{2} \ = \sqrt[12]{2} = 1.05946$$

On peut donc avoir, par exemple, la fréquence du La# 3 :

$$La\#\ 3 = La\ 3\ \times r = 440\ \times 1.05946 \cong 466.16240$$

Vérification de la valeur du cent :

$$c = 12\ \times \log_2(r) = 12\ \times\ log_2(1.05946) = 1$$

Pour définir notre gamme chromatique, on peut l'écrire comme une suite de nombres en cents :
0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100

Ce qui correspond bien à la suite de nombres qui est stockée par défaut comme **Tuning.ET12** dans FoxDot.

Des études poussées en théorie de la musique ont amené à créer et utiliser des tempéraments dont l'unité cyclique de référence serait différente de l'octave (tritave, quinte, …), ou bien dont le nombre de notes constituant l'unité cyclique serait différent de 12. Des exemples sont le tuning de Bohlen-Pierce (division du tritave en 13) ou le 17-EDO (division de l'octave en 17). Par ailleurs certains tunings historiques et culturels, comme ceux que l'on peut trouver dans la musique Indienne ou Balinaise, sont totalement différents du 12-EDO occidental.

En pratique les tunings peuvent aussi être définis par des suites de nombres :

```
# FoxDot permet d'automatiser le calcul sans renseigner les valeurs à la
main, et dispose aussi de quelques tunings :
#17-EDO :
print(P[:18]*12/17)

# Bohlen-Pierce
print(Tuning.bohlen_pierce)

# Intonation juste
print(Tuning.just)
```

B/ Les gammes, la tonalité et les notes

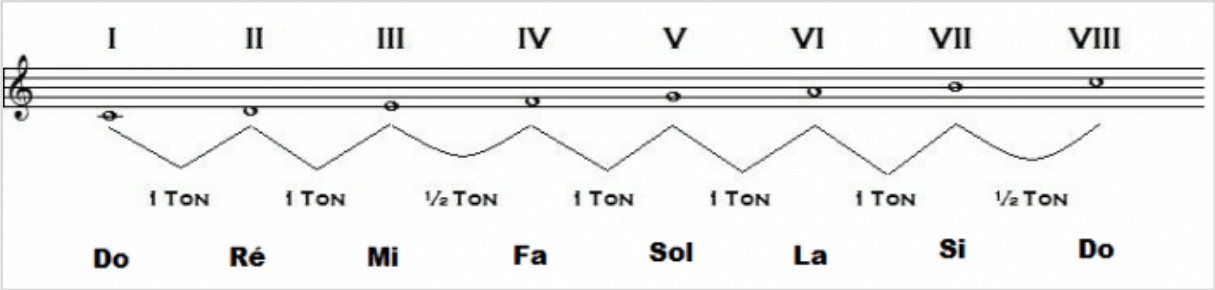
[AN_TONALITE]

En solfège on dit qu'il existe plusieurs gammes (ou échelles) et modes, tous ayant des implications techniques et historiques. Les plus connus sont les gammes mineure et majeure.

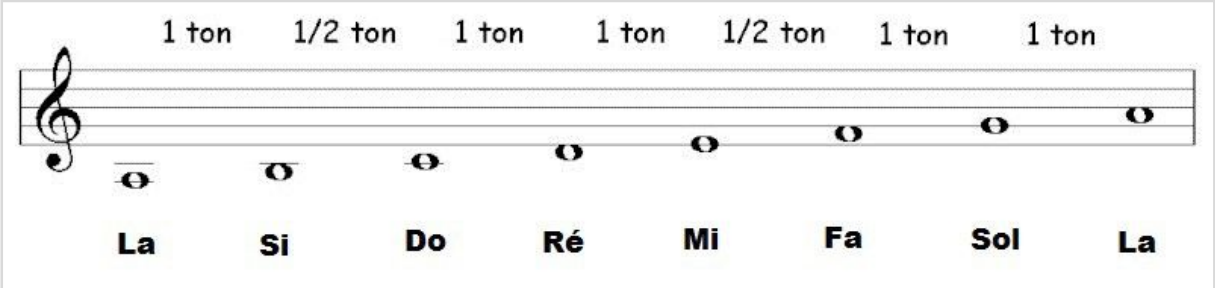
Lorsqu'une échelle unique est utilisée lors d'une chanson entière, on dit qu'elle est sa tonalité. Pour définir une tonalité sur une partition, on en place l'armure au début de la partition.

Le cycle des quintes permet de retrouver facilement la tonalité d'une chanson à partir du nombre de dièses ou de bémols à son armure.

Gamme majeure :



Gamme mineure :



Sur FoxDot, les gammes correspondent à des suites de nombres entiers.

```
# Vérification du tuning par défaut
print(Tuning.ET12) # Retourne [0,1,2,3,4,5,6,7,8,9,10,11]

# Gammes
print(Scale.chromatic) # Retourne [0,1,2,3,4,5,6,7,8,9,10,11], toutes les notes du tuning par défaut
print(Scale.major) # Retourne [0,2,4,5,7,9,11]
# On voit que l'échelle majeure est issue d'un filtre de sélection, qui supprime de la gamme chromatique, les notes n'obéissant pas aux règles de la gamme majeure.
```

La liste de chiffres contenue dans la gamme correspond aux indexes des notes présentes dans le tuning. On peut donc voir notre Scale.major comme un tableau associatif :

Index Scale.major	Valeur Scale.major	Valeur de la note dans le tuning	Signification si racine est Do
0	0	0	Do
1	2	2	Ré
2	4	4	Mi
3	5	5	Fa
4	7	7	Sol
5	9	9	La
6	11	11	Si

De la même manière, lorsqu'on crée un pattern de notes associé à une gamme, les notes utiliseront l'index des notes dans la gamme appropriée. Toujours pour une gamme de Do Majeur :

```
P[1,4,5.5,13] # signifie :
```

Note demandée	Valeur dans Scale.major	Signification
1	2	Ré
4	7	Sol
5.5	Moyenne pondérée entre 9 et 11 : 10	La#
13	11+2	Ré à l'octave au-dessus

On peut utiliser des nombres décimaux mais cela produira des notes en-dehors du tuning ou de la gamme définis : l'opération qui se produit est vraisemblablement que FoxDot cherchera à retourner une valeur quoiqu'il en soit, par exemple en opérant une moyenne pondérée. Donc une valeur de 5.5 ne produira pas le même son qu'une valeur de 5.6.

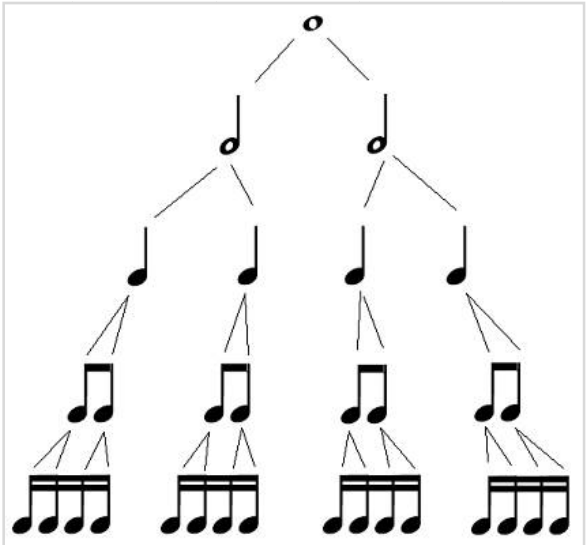
Atention :
L'échelle Scale.chromatic, ne fonctionne correctement que lorsqu'on utilise des tunings contenant 12 notes ! Pour utiliser d'autres tunings il faut redéfinir une échelle chromatique contenant le bon nombre de notes.

C/ Le rythme

[AN_RYTHM]

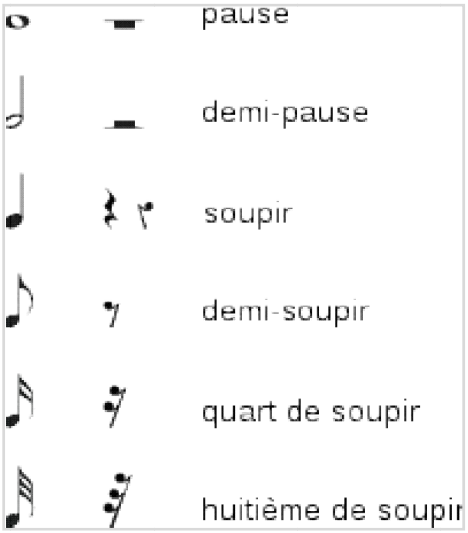
Le rythme s'organise en mesures sur une partition. Quelle que soit la signature de temps choisie; la note occupant le temps complet d'une mesure est appelée la **ronde**. Les notes issues des multiples subdivisions de la ronde, portent respectivement les noms de **blanche, noire, croche, double-croche, triple-croche, quadruple-croche**.

Hiérarchie des durées des notes :



Sur une partition, l'écriture d'un motif percussif se fait différemment de l'écriture des notes; en effet chaque composant de l'ensemble de batterie reçoit un symbole spécifique.

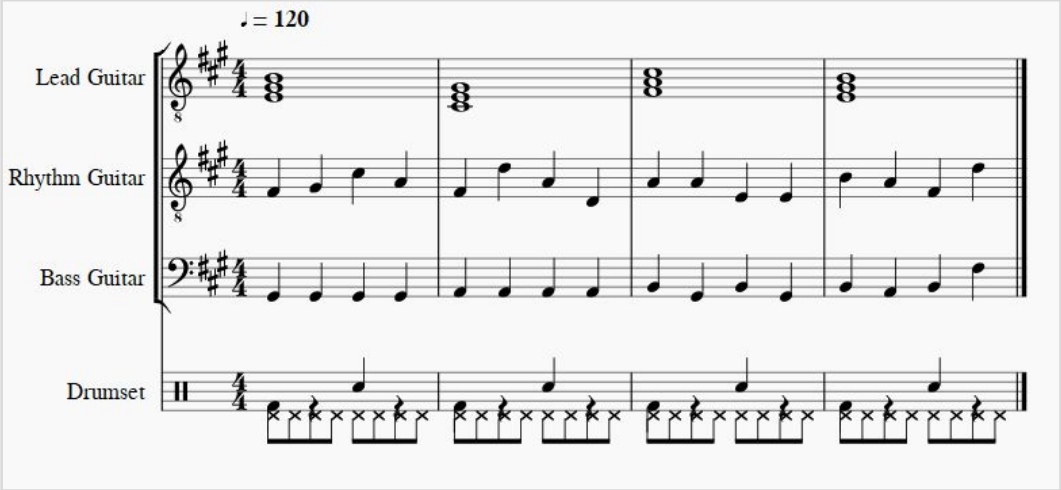
Les temps de silence ont aussi des notations particulières associées, toujours en fonction des subdivisions :



En 4/4; le tempo est associé à la durée d'une noire (un beat), en effet il représente le nombre de noires par minute.

Annexe 3 : Correspondance entre du code FoxDot et une partition

Par exemple la partition suivante : [AN_PART]



Il s'agit d'une partition de test, créée via le logiciel MuseScore, contenant une suite d'accords, une mélodie, une ligne de basse et un motif percussif.
On remarque le tempo 120, la signature de temps 4/4, ainsi que les 3 dièses, ce qui veut dire d'après le cycle des quintes, que nous sommes en tonalité de La Majeur ou Fa# mineur. Disons que l'on choisisse La Majeur : **La, Si, Do#, Ré, Mi, Fa#, Sol#, La**.

On lit les notes sur la clef de Sol et la clef de Fa. Attention : sur FoxDot, puisque l'on va définir notre racine à La, toute note se trouvant en-dessous prendra un nombre négatif.

Dans les percussions on remarque qu'on a des charlets, une grosse claire et une caisse claire.
Pour jouer un tel morceau sur FoxDot, voici la marche à suivre :

```
Clock.bpm = 120
Scale.default.set("major")
Root.default.set(10)

# accords
ac >> saw(P[-3,-5,-2,-3], dur=4) + P(0,3,5)

# mélodie mesure 1
me >> pluck(P[-2,-1,2,0])

# pattern à ré-écrire selon la mesure
me.degree=P[-2,3,0,-3] # mesure 2, et ainsi de suite

# ligne de basse – même procédé
ba >> dbass(-3) # pour la mesure 1, et ainsi de suite

# Percussions
pe >> play("<x o ><[--]>")
```

Il existe différents moyens de rendre plus rapide le changement de pattern désiré lorsqu'on joue la mélodie ou la ligne de basse.

Annexe 4 : Bien planifier l'exécution du morceau

[AN_PAT]

Comme nous l'avons vu, jouer un morceau en direct nécessite d'avoir une bonne connaissance des notes et des rythmes, car lorsqu'on part de zéro et qu'on commence l'enregistrement, c'est la même chose que quand on joue de la guitare ou du piano : il faut bien connaître quoi faire, et comment le faire.

Sur FoxDot, chaque piste se décompose en plusieurs parties auxquelles l'on doit penser :

- Le SynthDef utilisé
- L'enchaînement de notes de la mélodie
- La durée de chaque notes pour que l'ensemble soit cohérent
- La maîtrise des différents effets que l'on peut appliquer : distorsion, filtres, etc.

Ici attardons-nous sur les trois premiers, qui sont primordiaux pour que la chanson soit reconnaissable.

Le SynthDef à utiliser est un choix à faire en connaissance de cause. Certains Synthdefs, comme space, star, keys ou saw, sont mieux adaptés à la lecture d'accords; alors que d'autres sont utilisés pour les mélodies : karp, pluck, blip, sitar,… Pour se faire une idée, mieux vaut tous les tester.

Chaque player est traitée par l'objet Clock. La Clock ajoute les nouvelles requêtes à la file d'attente à chaque à chaque mesure.

La Clock est garant de l'exécution en temps réel de chaque requête en cours. Un player dans la liste de requêtes va être joué en boucle. On peut voir le contenu d'un player comme un tableau associatif, dont le nombre d'éléments est défini par l'attribut ayant la liste la plus longue. L'index 1 de la liste des degree va se combiner avec l'index 1 de la liste des dur, si une des deux listes arrive à extinction de manière anticipée, alors elle reprend au début.

Chaque attribut, y compris les effets, va donc s'appliquer à la note concernée.

Considérons les exemples en tonalité de Do Majeur et en signature 4/4 :

```
b1 >> dbass()
b2 >> dbass(dur=[1,1/2]
b3 >> pluck([1,2,5,3],dur=[1,1/2,3/2])
b4 >> pluck([1,2,5,3],dur=[1,1/4,1/8,2])
b5 >> pluck(P[1,2,5]*2,dur=PDur(3,8))
b6 >> blip(P[1,2,3,4],dur=2, hpf=[200,400])
```

L'instruction donnée par chaque player peut être vue comme une phrase :

	Instruction	P
b1	Jouer **noire Do** en boucle	4
b2	Jouer **noire Do** puis **croche Do**, en boucle	4
b3	Jouer **noire Ré**, puis **croche Mi**, puis **noire pointée La**, puis **noire Fa**, etc..	12
b4	Jouer **noire Ré**, puis **double croche Mi**, puis **triple croche La**, puis **blanche Fa**, etc…	108
b5	Jouer **croche pointée Mi**, puis **croche pointée Sol**, puis **croche Fa (octave sup.)**, etc…	12
b6	Jouer **blanche Ré (filtre passe-haut 200)**, puis **blanche Mi (hpf 400)**, puis **blanche Fa (hpf 200)**, puis **blanche Sol (hpf 400)**, etc…	8

La notion de périodicité (P) évalue au bout de combien de beats le motif demandé va retourner à la configuration exacte où il se trouvait à son départ. On voit par exemple que le player b3 a une période de 12 beats ((1+1/2+3/2) x 4 notes), alors que b1 et b2 ont respectivement des périodes de 1 et 2 beats. Comme 1 et 2 sont des diviseurs de 4, on peut extrapoler la durée à 4 beats (1 mesure). Si l'on joue b2 et b3 conjointement, il faudra attendre 3 mesures (ou 12 beats) avant d'entendre exactement la même combinaison de sons qu'en mesure 1.

De même, b4 va avoir une période différente :

$$1+\frac{1}{4}+\frac{1}{8}+2=\frac{8+2+1+16}{8}=\frac{27}{8}$$

Multiplié par 4 notes :

$$\frac{27\times 4}{8}=\frac{108}{8}=13.5$$

Multiplié par 8 : 108;
On doit multiplier par 8 car les produits par 1, 2 ou 4 ne donnent pas un nombre divisible par 4;
Il faut donc arriver à 108 beats (ou 27 mesures) pour que ce motif particulier revienne à sa configuration initiale.

Dans b5, PDur(3,8) a une durée totale de 2 beats. Multiplié par 3 notes cela donne une périodicité de 6 beats.
Pour extrapoler cette périodicité en nombre de mesures, on doit trouver un nombre multiple de 6 **et** de 4, le plus simple est 12, soit 3 mesures.

Attention :
Les exemples donnés ci-dessus ne sont vrais qu'en signature de temps 4/4, car dans d'autres configurations, le nombre de beats constituant une mesure va changer.
En ¾ il s'agit de 3 beats (une blanche et une noire), en 5/4 il s'agit de 5 beats (une blanche et une blanche pointée).

Sources

[SOURCES]

Installation et documentation de SuperCollider :
https://github.com/supercollider/supercollider
http://doc.sccode.org/Guides/Tour_of_UGens.html
http://doc.sccode.org/Tutorials/Getting-Started/00-Getting-Started-With-SC.html

FoxDot officiel :
http://foxdot.org
http://foxdot.org/installation/
https://docs.foxdot.org/

Fichiers python contenant une démonstration de tous les items documentés
https://github.com/Qirky/FoxDot-Worksheet/blob/master/worksheet.py
Fichiers python des tutorials
https://github.com/Qirky/FoxDot/tree/master/FoxDot/demo
Fichiers SuperCollider de tous les SynthDefs
https://github.com/Qirky/FoxDot/tree/master/FoxDot/osc/scsyndef

Interview de Ryan Kirkbride :
https://www.blog.pythonlibrary.org/2018/07/09/pydev-of-the-week-ryan-kirkbride/

Chaînes youtube de Ryan Kirkbride :
Guides FoxDot : https://www.youtube.com/channel/UCRyrNX07lFcfRSymZEWwl6w
Compositions de Ryan : https://www.youtube.com/user/DynamiteBananas

Collectif officiel du Live Coding :
https://toplap.org/about

Généralités sur le live coding, SuperCollider et son utilisation avec ixi lang :
http://www.ixi-software.net/thor/ixilang.pdf

La musique, les cents et le tuning :
https://fr.wikipedia.org/wiki/Cent_et_savart
https://fr.wikipedia.org/wiki/La_440
https://www.deleze.name/marcel/physique/musique/Frequences.pdf
https://fr.wikipedia.org/wiki/Syst%C3%A8me_tonal
https://fr.wikipedia.org/wiki/Gamme_de_Bohlen%E2%80%93Pierce

Le solfège, les gammes, le cycle des quintes :
https://www.composersimplement.com/theorie-musicale/

Les signatures de temps :
https://fr.wikipedia.org/wiki/Mesure_%C3%A0_cinq_temps

Calculs en ligne :
https://www.dcode.fr/ppcm
https://www.calcprofi.fr/calculatrice-racine-nieme.html