Python pour ingénieur·e·s : Parsers
Fichiers pour synthétiseur.

1 Présentation générale

L'objectif de cette série d'exercice est de faciliter l'utilisation du synthétiseur que nous avons écrit lors de la séance 3. Pour cela, nous aimerions pouvoir décrire, dans un fichier, la musique à jouer, plutôt que de devoir appeler à la main les fonctions de synthèse. Nous allons donc définir un format de fichier et écrire un parser pour lire de tels fichiers, et appeler les fonctions en fonction du contenu.

Présentation du format de fichier. Nous aimerions que les fichiers puissent être écrits à la main, nous allons donc utiliser un format textuel, qui contiendra les notes à jouer. Dans un premier temps, nous pouvons penser à un fichier contenant seulement une suite de fréquences à jouer pendant un certain temps. Par exemple, un fichier contenant la chaine de caractères « `440, 5; 262, 3` » produirait un *la* (440Hz) pendant 5 secondes, puis un *do* (262Hz) pendant 3 secondes.

Pour se simplifier la vie, on aimerait pouvoir nommer les notes: plutôt que se souvenir qu'un *do* est à 262Hz, on aimerait pouvoir appeler « `C4` », en ayant préalablement défini « `C4` » comme étant 262Hz. Ainsi, notre format de fichier aura une première partie avec des définitions de notes, et une seconde partie où il y aura le morceau à jouer. Ce qui donnerait, par exemple:

```
A4 = 440
C3 = 262
A4, 5; C3, 3
```

Remarque : On utilise la notation anglo-saxonne pour le nom des notes : notées de « `A` » (*la*) à « `G` » (*sol*), suivi d'un chiffre qui indique l'octave de référence. Par exemple : « `A4` » est le *la* standard, à 440Hz; « `B4` » est le *si* un ton plus haut. Le *la* « `A5` » est une octave plus haut (donc 880Hz). Le *sol* « `G3` » est un ton en dessous de « `A4` ».

On veut aussi pouvoir superposer plusieurs notes simultanément. Donc on introduit la notation « `... | ...` » qui indique que les deux éléments qui précèdent et suivent doivent être joués en même temps. De fait, on va devoir parenthéser les suites de notes pour désambiguïser certaines expressions. Au final, nos fichiers ressemblerons à :

```
C3 = 262
E3 = 330
G3 = 392
{
 C3, 5;
 {
  C3, 5;
  |E3, 5;
  |G3, 5;
 }
}
```

En l'occurrence, ce fichier joue un *do* pendant 5 secondes, puis un accord *do*/*mi*/*sol* pendant 5 secondes.

2 Format de fichier

Après ce que nous avons vu plus haut, nous pouvons maintenant écrire formellement la grammaire des fichiers que nous lirons :

```
fichier   := note_def* piste
note_def  := nom '=' freq
nom       := majuscule chiffre
majuscule := 0x41 ... 0x5A            ; 'A' ... 'Z'
chiffre   := 0x30 ... 0x39            ; '1' ... '9'
freq      := nombre
piste     := note | '{' piste* '}' | '{' piste ('|' piste)* '}'
note      := nom ',' nombre ';'
nombre    := chiffre*
```

Parser

Pour nous familiariser avec la grammaire, essayez de faire correspondre les groupes à la chaine ci-dessous :
`C3 = 262 E3 = 330 G3 = 392 { C3, 5; { C3, 5; | E3, 5; | G3, 5; } }`

3 Structure de données

3.1 Structure de données pour les pistes

Avant d'attaquer à écrire le parser, on va réfléchir à *ce qu'on veut obtenir*, comme sortie du parser. On aimerait avoir un objet que l'on puisse *synthétiser*, à l'aide du synthétiseur qu'on a écrit (par exemple à l'aide d'une méthode `synthetise`). Par ailleurs, on note que la structure de la piste est récursive : par exemple, pour synthétiser plusieurs pistes en parallèle, on pourra synthétiser chaque piste, puis les composer à l'aide de la fonction `superposer_signaux` que l'on a écrit dans le synthétiseur.

Ainsi, on peut imaginer avoir 3 classes, car une piste peut avoir trois formes : ou bien une note unique (modélisé par une classe `Son`), une séquence de pistes (modélisée par une classe `Sequence`), et une superposition de son (modélisée par une classe `Superposition`). Les classes `Sequence` et `Superposition` contiendront une liste de pistes (chacune pouvant être un son individuel, ou (récursivement) une séquence, ou une superposition).

Synthétiser les `Sequences` ou les `Superpositions` est (techniquement) facile[1]. En revanche, il y a une question pour ce qui est des notes de base. Notre classe `Son` a accès au nom de la note à jouer et à la durée, mais il faut convertir le nom de la note en sa fréquence, pour pouvoir générer le signal à l'aide des fonctions `generer_signal` ou `note` du synthétiseur. Pour cela, il faut retrouver, parmi toutes les notes définies, laquelle correspond. On peut donc simplement convertir l'ensemble des définitions en un dictionnaire des `noms` vers les `fréquences`, puis passer ce dictionnaire lors de la synthèse.

Représentation mémoire

Faites un schéma de la structure de données qui correspond à la chaine de caractère ci-dessous :
`C3 = 262 E3 = 330 G3 = 392 { C3, 5; { C3, 5; | E3, 5; | G3, 5; } }`

Le squelette des classes `Son`, `Sequence` et `Superposition` sont fournies.

[1] L'intuition peut toutefois être difficile à saisir, en raison de la récursion.

Implémentation de la fonction de synthèse

Complétez les fonctions `synthesize` des trois classes.

Remarque : Pour vous entrainer et pour bien comprendre le fonctionnement de ces classes, vous pouvez construire une piste à la main, puis tenter de la synthétiser.

3.2 Structure de données pour la définition des notes

Il faut aussi réfléchir à la manière de stocker les notes définies. Comme dit plus haut, on aimerait avoir un dictionnaire des noms de notes vers leur fréquence. Ce dictionnaire sera construit à partir d'une liste de 2-uplets correspondant aux couples définis.

4 Implémentation du Parser

4.1 Tokenizer

Un tokenizer partiel est fourni, mais il ne tokenise pas les lettres et les chiffres, ni les blancs (espaces, tabulations, saut de lignes, etc.).

Tokenizer

Complétez la spécification du tokenizer afin de lire les lettres (majuscules), les chiffres et les blancs.

Tokenization

Que renvoie la fonction `_tokenize("A4 = 440 C3 = 262 A4, 5; C3, 3")` ?

Remarque : Utilisez le debugger pour explorer l'exécution du programme.

4.2 Parser

Le parser doit transformer la sortie du tokenizer en une structure de donnée utilisable. Nous allons procéder en deux étapes : d'abord, reconnaitre les séquences de tokens qui nous intéressent, à l'aide des combinateurs de `funcparserlib`. Le résultat de ces combinaison est sous forme de type de base python (tuples, listes, etc.). Dans un second temps, nous transformerons donc ces objets renvoyés en objets qui nous intéressent (des `Sons`, des `Sequences` et des `Superpositions`).

La construction de parser par combinateurs fonctionne en construisant de petits parsers très simples, puis en les combinants jusqu'à obtenir ce qu'on souhaite. Quelques uns de ces petits parsers sont déjà fournis.

Observation des parsers fournis. Observez le parser `nom` :

```
nom = tok("majuscule") + tok("chiffre") >> (lambda l: ''.join(l))
```

Ceci créé un parser qui va chercher un token "majuscule", puis un token "chiffre". La séquence de ces deux tokens est exprimée à l'aide de la composition +[2]. L'opérateur + prend deux parsers (dans notre cas, le parser de majuscule et le parser de chiffre) et renvoie un nouveau parser. Au moment de parser une chaine de caractère, ce nouveau parser renvoie un tuple où le premier élément est le résultat du premier parser, et le second celui du second parser[3].

La seconde partie de la ligne (le `>> (...)`) ne nous intéresse pas pour le moment, on verra à quoi il sert un peu plus bas.

[2] La bibliothèque `funcparserlib` redéfini certains opérateurs, par exemple le +, qui n'est donc pas l'addition, mais bien la composition en séquence.

[3] Ce qui précède n'est pas tout à fait vrai : on peut combiner n parser en séquence, et au lieu d'obtenir des 2-uplets imbriqués, on obtient un seul n-uplet.

Typage de combinateurs

D'un point de vue typage, un parser est une fonction qui accepte une séquence de tokens, et qui renvoie un objet d'un type donné, et la suite de la séquence non consommée : `p: List(A) → B×List(A)`.

Écrivez le type du combinateur `+`, celui du combinateur `many`, celui du combinateur `-` ?

Premiers pas vers les combinateurs. On va maintenant écrire nos premiers parsers.

Écriture de parsers

Complétez les parsers `note_def` (qui doit accepter une définition de note) et `son` (qui doit accepter une note unique).
Pour le moment, on veut juste accepter les entrées correctes, on ne se préoccupe pas de l'objet renvoyé.

Remarque : Vous pouvez tester votre parser en appelant (par exemple) : `node_def.parse("...")`

Transformer la sortie d'un parser. On veut maintenant transformer la sortie des parsers que l'on a écrit. Par exemple, on aimerait construire un objet `Son` à partir du résultat du parser `son`.

Pour cela, on va utiliser un combinateur particulier : `>>`. Ce combinateur prend un parser `p` qui renvoie un objet de type `A` et une fonction `f` qui transforme `A` en `B` (donc une fonction `f: A → B`), et renvoie un parser qui parser à l'aide de `p`, puis applique `f` à la sortie, renvoyant finalement un objet de type `B`.

Transformation de parser

Écrivez une fonction `vers_note_def` qui prend un seul argument. Cet argument est un 2-uplet, donc le premier élément est un nom de note, et le second un entier (qui est une fréquence). Cette fonction doit renvoyer un 2-uplet contenant le nom de la note en 1ère position, et la fréquence en 2nde position.

Dans le parser `note_def` que vous avez écrit plus haut, ajoutez `>> note_def` à la fin de la ligne. Observez au débugger comme le résultat renvoyé par le parser change.

Remarque : Assurez vous que le parser ignore le token `=`, à l'aide du combinateur `-`, ou alors modifiez votre fonction `vers_note_def` afin d'avoir un 3-uplet comme argument.

Fonctions anonymes

Pour les fonctions courtes et qu'on ne réutilise pas, plutôt que de créer une fonction séparée, il est possible de créer des fonctions anonymes. Le format est:

```
lambda args: expression
```

Par exemple, l'expression suivante créé une fonction anonyme a deux arguments, et renvoie leur somme :

```
>>> lambda x, y: x+y
<function <lambda> at 0x76819dc23240>
```

On peut ensuite appeler la fonction comme on appellerait une fonction nommée :

```
>>> (lambda x, y: x+y)(2, 3)
5
```

Cette notation est assez pratique pour transformer les sorties de parser, puisqu'on veut souvent juste appeler un constructeur avec les valeurs de retour de chacun des sous-parsers. Si toutefois vous n'êtes pas à l'aise avec la notation, vous pouvez toujours passer par des fonctions classiques.

4.3 Parser récursif

Nous allons maintenant nous attaquer le dernier morceau du parser : la piste à jouer en elle-même. La difficulté, par rapport à ce qui précède, est qu'une piste est composée ou bien d'une note unique, d'une séquence de pistes, ou de pistes en parallèle : la définition est récursive[4].

Le problème, c'est qu'on ne peut pas écrire directement : en effet, si on écrit `piste = ... piste ...`, le nom `piste` à droite du = n'est pas encore défini (la variable n'existe pas encore).

Pour résoudre ce problème, on va prédéclarer la variable, puis définir le parser dans un second temps :

```
piste = forward_decl()
piste.define(... comme habituellement ...)
```

Parser de pistes

Définissez le parser `piste`.

Remarque : Procédez par étapes. Vous pouvez commencer par juste autoriser les notes individuelles, puis les notes et les séquences de pistes, puis tout.

Pensez à convertir la sortie de chaque sous-parser en un objet adéquat : `Son`, `Sequence`, ou `Superposition`.

Remarque : Il peut arriver, si vous ne faites pas attention, que vous ayez des récursions infinies (typiquement, si on a un parser défini par `piste.define(piste)`). Si c'est le cas, assurez-vous de bien *consommer* des tokens à chaque appel récursif (normalement, chaque appel récursif consomme au moins une paire d'accolades).

Test final

Vous devriez maintenant pouvoir tester votre parser sur un fichier réel. Vous pouvez décommenter les dernières lignes de votre code source pour synthétiser le morceau écrit dans `musique.txt`.

[4] Cette récursivité est très pratique : par exemple, on peut écrire un accord à l'aide de plusieurs notes en parallèles, qui prend place dans la ligne de main droite d'un piano (donc une séquence de notes), qui est elle-même en parallèle avec la main gauche.