Programmation orientée objet en python

1. Introduction

La programmation orientée objet est le mode de programmation fondamental dans les langages Java et C++, deux langages utilisés pour la réalisation de logiciels complexes. Elle peut aussi être pratiquée dans les langages PHP et javascript.

Elle est au cœur du langage Python. Par exemple, les listes sont des objets de la classe `List`. Ce document constitue une introduction à la programmation orientée objet en Python. On se propose d'appliquer les méthodes abordées sur un exemple : le traitement numérique du signal. L'exemple traité comportera deux types d'objets : les signaux et les filtres.

On aura besoin des modules suivants :

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import firwin
import wave
```

2. Classe et instance

Un objet appartient à une classe. Une classe comporte des attributs. Les attributs sont des données ou des fonctions qui traitent ces données, appelées aussi méthodes. Nous allons créer une classe pour les signaux :

```
class Signal:
    N = 0 # nombre d'échantillons
    fechant = 0 # fréquence d'échantillonnage
    x = np.zeros(N) # tableau des échantillons
    type = 'aucun'
    def __init__(self,N,fechant): # initialisateur
        self.N = N
        self.fechant = fechant
        self.x = np.zeros(N)
    def tracer(self): # tracer le signal avec matplotlib
        pass
    def enregistrer(self,fichier): # enregistrer le signal dans un fichier texte
        pass
        # utiliser numpy.savetxt
    def lire(self,fichier): # lire le signal dans un fichier texte
        pass
        # utiliser numpy.loadtxt
    def dupliquer(self): # renvoyer une copie du signal
        pass
        # renvoyer un nouvel objet de la classe Signal
    def filtrer(self,filtre):
        pass
        # filtrer avec un objet de la classe Filtre (définie plus loin)
        # renvoie un nouveau signal
```

Une classe se déclare avec le mot-clé `class`. Le nom de la classe est ici `Signal`. Cette classe comporte 4 attributs de type données et 6 méthodes. La première méthode, nommée `__init__` est l'initialisateur d'instance. Cette méthode est appelée lorsqu'on crée un objet de la classe, appelé aussi *instance de la classe*. Dans le cas présent, cette méthode a deux paramètres (en plus du paramètre nommé `self`). Dans cet exemple, la déclaration des trois attributs de classe (`N,fechant,x`) au début de la définition de la classe aurait pu être omise car l'initialisateur crée de toute façon trois d'attributs d'instance du même nom. À la différence des attributs de classe, les contenus des attributs d'instance peuvent être différents d'une instance à l'autre. Par exemple, dans le cas présent, deux instances différentes pourront avoir deux valeurs différentes de `N`. La déclaration d'attributs de données au niveau de la classe a surtout un intérêt pour des attributs qui doivent avoir une valeur par défaut et qui ne sont pas réaffectés par l'initialisateur, par exemple l'attribut nommé `type` dans cet exemple. Le fait d'avoir déclaré `N` comme attribut de classe n'a d'intérêt que pour la clarté du code, car l'appel automatique de l'initialisateur crée un attribut d'instance qui porte le même nom, ce qui fait disparaître l'attribut de classe.

Voici comment créer une instance :

```
signal_1 = Signal(1000,10e3) # signal à 1000 échantillons, fréq. d'échantillonnage 10 kH
```

Une deuxième instance de cette classe est créée par :

```
signal_2 = Signal(1000,10e3) # signal à 1000 échantillons, fréq. d'échantillonnage 10 kH
```

Ces deux instances ont la même structure mais les données contenues dans les attributs (d'instance) de l'une sont complètement indépendantes des données des attributs de l'autre.

Une classe peut être utilisée comme une simple structure de données. Pour modifier un attribut, il suffit par exemple d'écrire :

```
signal_1.N = 2000 # à proscrire
```

On constate cependant que ce changement conduit à une incohérence dans les données, puisque le tableau contenant les échantillons n'a plus la taille correspondant à l'attribut `N`. Pour éviter ce type d'erreur, la programmation orientée objet se fait en suivant le principe de *l'encapsulation des données* : les attributs ne doivent être modifiés que par une méthode de la classe. En effet, seul le concepteur de la classe sait comment les différents attributs doivent être modifiés lorsqu'on change des données. Dans cet exemple, la modification de la taille du signal imposerait de redéfinir aussi le tableau contenant les échantillons.

Le premier paramètres des méthodes de la classe contient l'instance de la classe sur laquelle la méthode doit agir. Il est d'usage de nommer cet argument `self` mais tout autre nom est permis, par exemple `soi` si l'on préfère le français. Par exemple pour l'initialisateur, `self` désigne l'objet qui est en train d'être créé. Pour appeler une méthode sur une instance, on utilise la syntaxe suivante :

```
signal_1.enregistrer("signal_1.txt")
```

Cette instruction a pour effet d'appeler la méthode `enregistrer` de la classe `Signal` avec l'instance `signal_1` comme premier argument (l'argument nommé `self`). Elle est équivalente à l'instruction suivante :

```
Signal.enregistrer(signal_1,"signal_1.txt")
```

On utilise toujours la première syntaxe, plus concise et qui dispense de donner le nom de la classe, d'autant plus que, comme nous le verrons plus loin, deux objets peuvent comporter la même méthode sans appartenir exactement à la même classe.

3. Héritage

La classe `Signal` déclarée précédemment représente le cas le plus général de signal. Les 5 méthodes déclarées (qui restent à implémenter) sont en effet les mêmes pour tous les types de signaux. Pour définir un signal sinusoïdal, nous déclarons une classe `SignalSinus` qui hérite de la classe `Signal`. Cette classe représente un signal particulier, le signal sinusoïdal.

```
class SignalSinus(Signal):
    amplitude = 0
    frequence = 0
    def __init__(self,N,fechant,amplitude,frequence):
        Signal.__init__(self,N,fechant)
        self.type = 'sinus'
        self.amplitude = amplitude
        self.frequence = frequence
        #générer le signal sinusoïdal
    def afficher_frequence(self):
        print(self.frequence)
```

Cette classe hérite de tous les attributs de sa classe mère. Elle a en plus deux attributs de données (l'amplitude et la fréquence du signal sinusoïdal). Une classe héritée d'une autre peut contenir des méthodes qui ne sont pas dans la classe mère, la méthode `afficher_frequence` dans cet exemple. Toute méthode de la classe mère peut être redéfinie si son implémentation doit être modifiée. Dans le cas présent, la seule méthode redéfinie est l'initialisateur. En effet, il comporte deux arguments de plus et la génération du signal sinusoïdal doit se faire dans l'initialisateur. La première ligne de l'initialisateur comporte un appel à l'initialisateur de la classe mère. La méthode `afficher_frequence` a été ajoutée dans cette classe. Cette méthode ne pouvait pas exister dans la classe `Signal` puisque la fréquence n'est pas définie pour un signal en général.

Voici comment créer une instance de cette classe :

```
signal_sinus_1 = SignalSinus(1000,1e3,1,10)
```

Cette instance appartient à la classe `SignalSinus` mais aussi, par héritage, à la classe `Signal`. Toutes les méthodes de cette dernière sont ainsi utilisables, par exemple :

```
signal_sinus_1.tracer()
```

Il faut remarquer que l'appel de cette fonction ne nécessite pas de connaître la classe héritée à laquelle appartient l'objet. Il suffit de savoir que cette classe hérite de la classe `Signal`.

Pour définir des signaux de forme carrée, nous déclarons la classe suivante :

```
class SignalCarre(Signal):
    amplitude = 0
    frequence = 0
    def __init__(self,N,fechant,amplitude,frequence):
        Signal.__init__(self,N,fechant)
        self.type = 'carre'
        self.amplitude = amplitude
        self.frequence = frequence
        #générer le signal carré
    def afficher_frequence(self):
        print(self.frequence)
```

Nous constatons que la méthode `afficher_frequence` est identique dans les deux classes, puisque ces deux signaux sont périodiques. Dans ce cas, il serait judicieux de déclarer d'abord une classe `SignalPeriodique` héritée de la classe `Signal`, qui comporterait tous les attributs propres aux signaux périodiques, puis de déclarer deux classes `SignalSinus` et `SignalCarre` héritées de `SignalPeriodique`. L'héritage permet de définir des classes de plus en plus spécialisées, sans être obligé de réécrire les fonctions identiques.

La classe suivante permettra de lire le signal dans un fichier WAV (fichier audio sans compression) et de l'enregistrer à nouveau dans un fichier WAV après le filtrage :

```
class SignalWave(Signal):
    def __init__(self,fichier):
        pass
        # lecture du fichier wave avec le module wave
    def enregistrer_wave(self,fichier):
        pass
```

4. Une autre classe

Nous avons besoin d'une classe permettant de définir et d'implémenter un filtre numérique. Dans sa forme la plus générale, un filtre numérique est défini par la relation de récurrence suivante :

$$a_0 y_n + a_1 y_{n-1} + a_2 y_{n-2} + \ldots = b_0 x_n + b_1 x_{n-1} + b_2 x_{n-2} + \ldots$$

où x_n sont les échantillons du signal d'entrée et y_n ceux du signal de sortie.

Voici la classe générale pour un filtre :

```
class Filtre:
    na = 0
    nb = 0
    a = np.zeros(na)
    b = np.zeros(nb)
    type = 'aucun'
    def __init__(self,a,b):
        self.na = len(a)
        self.nb = len(b)
        self.a = np.array(a)
        self.b = np.array(b)
    def tracer(self):
        pass
        # tracé de la réponse fréquentielle
    def appliquer(self,x,y,n):
        pass
        # appliquer le filtre à l'indice n
        # x : signal d'entrée (tableau numpy.ndarray)
        # y : signal de sortie (tableau numpy.ndarray)
    def donner_taille(self):
        return (self.na,self.nb)
```

Cette classe ne comporte aucune définition concrète de filtre mais elle contient le tracé de la réponse fréquentielle et l'implémentation du filtrage, qui sont deux fonctions indépendantes des valeurs des coefficients du filtre.

Pour définir un filtre passe-bas convolutif, nous déclarons une classe héritée de la précédente :

```
class FiltreConvolutionPasseBas(Filtre):
    fc = 0 # fréquence de coupure divisée par fechant
    P = 0 # nombre de coefficients = 2*P+1
    def __init__(self,fc,P):
        self.type = 'passe-bas'
        self.fc = fc
        self.P = P
        b = firwin(2*P+1,cutoff=[fc],window='hann',nyq=0.5)
        Filtre.__init__(self,[1],b)
```

L'initialisateur de cette classe prend en arguments la fréquence de coupure relative et le nombre de coefficients. Voici comment définir un filtre passe-bas :

```
filtre_1 = FiltreConvolutionPasseBas(0.1,10)
```

Il faut remarquer que créer des instances des classes `Signal` ou `Filtre`, bien que possible, n'a pas d'intérêt puisque ces instances ne pourraient être utilisées pleinement. Pour la première, il manque les échantillons du signal. Pour la seconde, il manque les coefficients du filtre.

Un objet de la classe `Filtre` est utilisé dans la méthode `filtrer` de la classe `Signal`. L'implémentation de cette fonction nécessite la connaissance du nombre de coefficients a_n et b_n du filtre. C'est pour cela que la classe `Filtre` comporte une méthode `donner_taille`. Bien qu'il soit en principe possible d'accéder directement aux attributs `na` et `nb` de la classe `Filtre`, le principe d'encapsulation des données exige de le faire uniquement par l'intermédiaire de méthodes. Supposons que le concepteur de la classe `Filtre` et de ses classes héritées décide de changer le nom d'un attribut de données. S'il ne change pas les entêtes des méthodes, un code qui utilise ces classes n'aura pas à être modifié à condition qu'il respecte bien l'encapsulation des données. En python, l'encapsulation des données n'est pas imposée par le langage mais elle est fortement recommandée. Dans la conception des gros logiciels, ce principe permet à différents développeurs de travailler sur différentes classes, une fois que les classes et les entêtes des méthodes sont fixés.

5. Solution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import firwin,freqz
import scipy.io.wavfile as wave


class Signal:
    N = 0 # nombre d'échantillons
    fechant = 0 # fréquence d'échantillonnage
    x = np.zeros(N) # tableau des échantillons
    type = 'aucun'
    def __init__(self,N,fechant): # initialisateur
        self.N = N
        self.fechant = fechant
        self.x = np.zeros(N)
    def tracer(self): # tracer le signal avec matplotlib
        plt.figure()
        t = np.arange(self.N)*1.0/self.fechant
        plt.plot(t,self.x)
        plt.xlabel("t (s)")
        plt.ylabel("x")
        plt.grid()
        plt.show()
    def enregistrer(self,fichier): # enregistrer le signal dans un fichier texte
        np.savetxt(fichier,np.concatenate(([self.N],[self.fechant],self.x)))
    def enregistrer_wave(self,fichier):
        wave.write(fichier,int(self.fechant),self.x.astype(np.float32))
    def lire(self,fichier): # lire le signal dans un fichier texte
```

```
        data = np.loadtxt(fichier)
        self.N = int(data[0])
        self.fechant = data[1]
        self.x = data[2:]
    def dupliquer(self): # renvoyer une copie du signal
        signal = Signal(self.N,self.fechant)
        signal.x = self.x.copy()
        return signal

    def filtrer(self,filtre):
        # filtrer avec un objet de la classe Filtre (définie plus loin)
        sortie = Signal(self.N,self.fechant)
        (na,nb) = filtre.donner_taille()
        debut = max(na,nb)
        for n in range(debut,self.N):
            filtre.appliquer(self.x,sortie.x,n)
        return sortie

class SignalSinus(Signal):
    amplitude = 0
    frequence = 0
    def __init__(self,N,fechant,amplitude,frequence):
        Signal.__init__(self,N,fechant)
        self.type = 'sinus'
        self.amplitude = amplitude
        self.frequence = frequence
        a = 2*np.pi*frequence/self.fechant
        for i in range(N):
            self.x[i] = amplitude*np.sin(a*i)
    def afficher_frequence(self):
        print(self.frequence)

class SignalCarre(Signal):
    amplitude = 0
    frequence = 0
    def __init__(self,N,fechant,amplitude,frequence):
        Signal.__init__(self,N,fechant)
        self.type = 'carre'
        self.amplitude = amplitude
        self.frequence = frequence
        a = 2*np.pi*frequence/self.fechant
        for i in range(N):
            if np.sin(a*i)>0:
                self.x[i] = amplitude
            else:
                self.x[i] = -amplitude
    def afficher_frequence(self):
        print(self.frequence)


class SignalWave(Signal):
    def __init__(self,fichier):
        fechant,data = wave.read(fichier)
        s = data.shape
        if len(s)==1:
            x = data
        else:
            x = data[:,0]
```

```
        Signal.__init__(self,x.size,fechant)
        self.x = x


class Filtre:
    na = 0
    nb = 0
    a = np.zeros(na)
    b = np.zeros(nb)
    type = 'aucun'
    def __init__(self,a,b):
        self.na = len(a)
        self.nb = len(b)
        self.a = np.array(a)
        self.b = np.array(b)
    def tracer(self):
        w,H=freqz(self.b,self.a)
        plt.figure()
        plt.subplot(211)
        plt.plot(w/(2*np.pi),np.absolute(H))
        plt.ylabel("G")
        plt.grid()
        plt.subplot(212)
        plt.plot(w/(2*np.pi),np.unwrap(np.angle(H)))
        plt.xlabel("f/fe")
        plt.ylabel("déphasage")
        plt.grid()
        plt.show()
    def appliquer(self,x,y,n):
        # appliquer le filtre à l'indice n
        # x : signal d'entrée (tableau numpy.ndarray)
        # y : signal de sortie (tableau numpy.ndarray)
        somme = 0
        for i in range(self.nb):
            somme += self.b[i]*x[n-i]
        for i in range(1,self.na):
            somme -= self.a[i]*y[n-i]
        y[n] = somme/self.a[0]
    def donner_taille(self):
        return (self.na,self.nb)


class FiltreConvolutionPasseBas(Filtre):
    fc = 0 # fréquence de coupure divisée par fechant
    P = 0 # nombre de coefficients = 2*P+1
    def __init__(self,fc,P):
        self.type = 'passe-bas'
        self.fc = fc
        self.P = P
        b = firwin(2*P+1,cutoff=[fc],window='hann',nyq=0.5)
        Filtre.__init__(self,[1],b)

class FiltreIntegrateur(Filtre):
    def __init__(self,g):
        self.type = "integrateur"
        Filtre.__init__(self,[1,-1],[g/2,g/2])

signal_1 = SignalCarre(10000,40000,1,200)
```

```
signal_1.enregistrer("signal1.txt")
signal_2 = Signal(0,0)
signal_2.lire("signal1.txt")

signal_1.tracer()
signal_3 = signal_2.dupliquer()
filtre_1 = FiltreConvolutionPasseBas(0.01,20)
filtre_1.tracer()
signal_4 = signal_1.filtrer(filtre_1)
signal_4.tracer()
filtre_2 = FiltreIntegrateur(1)
signal_5 = signal_1.filtrer(filtre_2)
signal_5.tracer()
signal_5.enregistrer_wave("signal_5.wav")
```