Chapitre 2.2
Fonctions et modules

1. Les fonctions

1.1 Pourquoi utiliser des fonctions ?

Lorsque l'on développe, on utilise énormément de fonctions, comme par exemple **`print`** ou **`input`**. Ces dernières sont assez simples à manipuler pour plusieurs raisons :

– elles portent un nom simple qui indique bien à quoi elles servent ;

– elles prennent des paramètres qui permettent de varier la manière dont on les utilise ;

– on n'a pas besoin de savoir comment elles sont écrites, juste ce qu'elles vont faire.

Lorsque vous écrivez votre propre code, vous allez devoir concevoir des algorithmes plus ou moins complexes, et lorsque vous n'êtes pas organisé, vous allez produire ce que nous avons fait jusqu'à présent : un code parfaitement linéaire.

L'inconvénient principal est le suivant : le code est une longue prose, sans repères particuliers. Il est difficile d'en isoler une partie et de toujours savoir quelle ligne est appelée à quel moment. Usuellement, on considère qu'une fonction bien faite doit faire une dizaine de lignes en moyenne, 20 à 25 au maximum.

Ces métriques ne sont pas, bien entendu, des obligations, mais un ordre d'idée à garder en tête et à essayer de respecter pour avoir un code lisible et compréhensible par tous, y compris par vous quelques mois plus tard, lorsque ce que vous avez écrit ne sera plus aussi frais qu'au moment où vous l'écrivez.

En effet, un code trop long est difficile à lire, à appréhender et donc à maintenir.

Le constat est donc clair, il faut organiser son code pour qu'il soit fait de petites briques simples et faciles à identifier. La réelle difficulté, c'est de savoir comment délimiter ces briques, comment en faire de belles fonctions qui soient suffisamment précises pour faire ce que vous souhaitez en détail, mais aussi suffisamment génériques pour ne pas avoir deux fonctions qui sont quasiment identiques et qui ne se distinguent que par un détail.

Un bon endroit pour commencer, c'est de regarder le code produit jusqu'à maintenant (la solution de l'exercice de la fin du chapitre précédent) et d'identifier des doublons dans le code :

```
print("Saisissez le nombre à deviner")
while True:
    nombre = input("Saisissez un nombre entre 0 et 99: ")
    try:
        nombre = int(nombre)
    except:
        pass
    else:
        if 0 <= nombre <= 99:
            break

# PARTIE 2
print("Essayez de trouver le nombre à deviner")
while True:  # BOUCLE 1
    while True:  # BOUCLE 2
        essai = input("Saisissez un nombre entre 0 et 99: ")
        try:
            essai = int(essai)
        except:
            pass
        else:
            if 0 <= essai <= 99:
                break  # Boucle 2

    if essai < nombre:
        print("Trop petit")
    elif essai > nombre:
        print("Trop grand")
    else:
        print("Gagné!")
        break  # Boucle 1
```

Remarque

Vous trouverez cet exemple dans le répertoire Guide du dépôt. Il se nomme 11_JEU_-guess_the_number.py.

On voit que dans cet extrait de code, demander la saisie du nombre qu'il faut trouver et celle d'un nombre à deviner est quasiment la même chose : il n'y a que le premier affichage qui indique à l'utilisateur ce que l'on attend de lui qui change.
Notez que l'élimination de doublons de code est quelque chose de très important, puisque lorsque vous devez maintenir un code, si vous devez changer quelque chose, il faut le répercuter sur tous les doublons et qu'il peut être assez aisé d'en manquer un dans l'opération.
Cet objectif d'élimination des doublons est donc l'endroit de notre fil rouge où l'on va commencer par définir notre première fonction utile.

Par contre, n'oubliez pas que dans la vraie vie, vous devez réfléchir d'abord et coder après et que, par conséquent, vous devez d'abord définir quelles briques vous allez créer avant de réellement les créer et non pas pondre un code d'abord et réfléchir à la manière de le rendre lisible après.

1.2 Introduction aux fonctions

1.2.1 Comment déclarer une fonction

En Python, les principes syntaxiques sont toujours les mêmes. Le code d'une fonction étant un bloc, la syntaxe d'une fonction est celle d'un bloc.
On va donc écrire le mot-clé **`def`** permettant d'indiquer que l'on définit une fonction, puis le nom de cette fonction, suivi de parenthèses (on verra plus tard ce que l'on peut y mettre) et le fameux deux-points. Cette première ligne est nommée la **signature de la fonction**.
Tout ce qui suit et qui est indenté est le **corps de la fonction**. Voyons ce que cela donne :

```
def demander_saisie_nombre():
    while True:
        saisie = input("Saisissez un nombre entre 0 et 99: ")
        try:
            saisie = int(saisie)
        except:
            pass
        else:
            if 9 <= saisie <= 99:
                break
    return saisie
```

À l'exception de la première et de la dernière ligne, l'ensemble de cet extrait de code est rigoureusement identique à la partie qui était en doublon dans notre première version du jeu.
La seule différence est le nom de la variable qui était **`nombre`**, puis **`essai`**, et qui est maintenant **`saisie`**.
En effet, le nom des variables correspondait, dans le programme de départ, respectivement au nombre à deviner puis à l'essai du joueur.
Ici, on est dans une fonction qui a simplement pour but de demander la saisie d'un nombre quelconque. Au niveau de la fonction, on ne sait pas à quoi ce nombre va servir, on ne sait pas non plus son nom et on n'a pas besoin de le savoir. On sait juste qu'il s'agit d'une saisie, on décide donc de la nommer ainsi.
Là où les choses deviennent intéressantes, c'est lorsque l'on va utiliser notre fonction :

```
# PARTIE 1
print("Saisissez le nombre à deviner")
nombre = demander_saisie_nombre()

# PARTIE 2
print("Essayez de trouver le nombre à deviner")
while True:
    essai = demander_saisie_nombre()
```

```
    if essai < nombre:
        print("Trop petit")
    elif essai > nombre:
        print("Trop grand")
    else:
        print "Gagné!"
        break
```

On note que le code est considérablement plus court et que l'on retrouve bien nos variables **nombre** et **essai**, en tant que résultat de la fonction.
C'est parce que la fonction a renvoyé la saisie que l'on peut réaliser cette affectation : vous comprenez maintenant le sens de l'instruction **return** à la fin de la fonction.

Remarque

Vous trouverez cet exemple dans le répertoire Guide du dépôt. Il se nomme 12_Fonctions.py.

Nous avons donc écrit notre première fonction et le programme se comporte, de notre point de vue d'utilisateur, exactement de la même manière.

1.2.2 Gestion d'un paramètre

Nous pouvons assez aisément améliorer notre fonction. En effet, au lieu de faire des affichages avant d'appeler notre fonction, nous pouvons faire en sorte de changer l'invite lorsqu'on nous demande une saisie.
Pour cela, il faut donc passer un paramètre, c'est-à-dire que celui qui appelle la fonction doit lui donner les éléments pour qu'elle puisse agir conformément à son souhait.
Dans notre exemple, nous souhaitons également définir les valeurs minimale et maximale une fois pour toutes : nous allons utiliser des constantes.
Cette notion est très importante, puisqu'en utilisant cette constante plutôt qu'un littéral, nous nous donnons les moyens de changer cette valeur simplement : il suffit de modifier la constante plutôt que de parcourir tout le code à la recherche d'un littéral à modifier.
Ainsi, une constante se définit exactement comme une variable, sauf qu'elle est en majuscules :

```
MIN = 0
MAX = 99
```

Pour Python, une constante est une variable comme une autre. Seule la convention qui consiste à les mettre en majuscules en fait des constantes, mais vous pouvez toujours les modifier, rien ne vous en empêche.

Remarque

Python fonctionne énormément avec des conventions : il vous donne les outils pour faire les choses correctement, mais il ne vous contraint pas. Python part du principe que le développeur sait ce qu'il fait et il vous fait entièrement confiance pour faire la bonne chose : si pour une raison ou une autre vous ne respectez pas la convention, c'est que vous avez une bonne raison de ne pas le faire et Python respecte cela.

L'utilisation de ces constantes augmente aussi la lisibilité et la compréhension du code, puisque, lors de la déclaration, on comprend tout à fait de quoi il s'agit et que si on rencontre **MIN** ou **MAX** plus loin dans le code, on comprendra ce à quoi cela réfère, plus simplement que s'il s'agissait de littéraux.
Voici pour commencer la fonction légèrement retravaillée :

```
def demander_saisie_nombre(invite):
    # Compléter l'invite:
    invite += " entre " + str(MIN) + " et " + str(MAX) + ": "

    while True:
        saisie = input(invite)
        try:
            saisie = int(saisie)
        except:
            pass
        else:
            if MIN <= saisie <= MAX:
                break
    return saisie
```

On se donne donc la possibilité d'appeler notre fonction en disant ce qu'il faut saisir, et on complète cette information en précisant les bornes du nombre à saisir.
Notez que les constantes **MIN** et **MAX** sont définies en dehors de la fonction. Pour autant, elles sont accessibles. C'est aussi le cas de toutes les variables qui sont définies, au moment où la fonction est appelée.

Remarque

Sauf cas exceptionnel et que l'on est certain de maîtriser, on évitera d'utiliser dans une fonction une variable qui pourrait ne pas être définie au moment où la fonction est appelée.

Si on y réfléchit bien, les fonctions **int** et **input** sont également définies en dehors de la fonction, et si on avait importé un module au début du fichier, il serait aussi accessible depuis l'intérieur de la fonction.
Si on veut aller plus loin sur ces questions, il faudra se reporter au chapitre Déclarations - section Visibilité, traitant de la **portée d'une variable**.
Voici le code qui utilise cette fonction :

```
# PARTIE 1
nombre = demander_saisie_nombre("Saisissez le nombre à deviner")

# PARTIE 2
while True:
    essai = demander_saisie_nombre("Devinez le nombre")
    if essai < nombre:
        print("Trop petit")
    elif essai > nombre:
        print("Trop grand")
```

```
    else:
        print("Gagné!")
        break
```

Le code ainsi écrit est beaucoup plus lisible : on sait tout de suite pourquoi on utilise notre fonction et ce qu'elle va produire.

Remarque

Vous trouverez cet exemple dans le répertoire Guide du dépôt. Il se nomme 13_Fonctions_génériques_1.py.

Nous allons voir maintenant comment cette fonction peut être utilisée encore plus intelligemment.

1.2.3 Comment rendre une fonction plus générique

La fonction telle que nous l'avons écrite dépend de **MIN** et de **MAX**. Si nous souhaitons que ces deux valeurs puissent varier, il nous faut ne plus utiliser des constantes. Mais la fonction elle-même ne sait pas de quelle manière ces deux valeurs peuvent varier.
Ces valeurs doivent donc devenir des paramètres :

```
def demander_saisie_nombre(invite, minimum, maximum):
    invite += " entre " + str(minimum) + " et " +
              str(maximum) + " : "

    while True:
        saisie = input(invite)
        try:
            saisie = int(saisie)
        except:
            pass
        else:
            if minimum <= saisie <= maximum:
                break
    return saisie
```

On voit donc apparaître deux nouveaux paramètres, **minimum** et **maximum** et, par rapport à l'exemple précédent, on a remplacé **MIN** par **minimum** et **MAX** par **maximum**, tout simplement.

Astuce

Notez au passage la continuation de ligne entre les lignes 2 et 3 : comme la ligne 2 se termine par un +, Python sait que la ligne 3 est la suite de l'instruction débutée ligne 2. Par convention, comme cette instruction est une affectation, on aligne la ligne 3 sur le début de l'opérande de droite.

Chapitre 5
L'ajout de sons dans un jeu Pygame

1. Introduction

Animer un jeu vidéo, c'est d'abord mettre en place le visuel et donner le mouvement à certains objets graphiques. Mais cela consiste également à ajouter du son au jeu, par exemple un fond sonore, une musique, pour ainsi améliorer l'expérience utilisateur. Il y a dans Pygame un module pour cela. Son étude et sa mise en œuvre constituent l'essentiel du présent chapitre.

Le module qui permet la gestion des sons en Pygame s'appelle *pygame.mixer.* Il contient deux grandes notions :

- Le sous-module *music* qui gère la musique de fond. Il n'y en a qu'une à la fois.
- L'objet *Sound* de *mixer* que l'on peut instancier plusieurs fois pour s'en servir par exemple pour les effets sonores du jeu.

2. La gestion du son avec Pygame

En premier lieu, il est nécessaire d'initialiser ce module grâce à la fonction *pygame.mixer.init.* Cet appel doit être fait explicitement. Le module *mixer* inclut un certain nombre d'outils relatifs aux effets sonores grâce à la classe *Sound.* Ce module inclut également une sorte de sous-module, *music*, dévolu à la gestion du fond sonore.

2.1 Les modules pygame.mixer et pygame.mixer.music

Remarque

Une documentation des modules *mixer* et *music* de Pygame est disponible dans le chapitre Les principaux modules Pygame.

2.1.1 Le module pygame.mixer.music (fond sonore)

Ce module Pygame permet de gérer le fond sonore (unique). Ci-après les principales fonctions disponibles dans *music*.

- *pygame.mixer.music.load* permet de charger un fichier de fond sonore.
- *pygame.mixer.music.play* permet de jouer/lire le fond sonore.
- *pygame.mixer.music.rewind* permet de reprendre au début le fond sonore.
- *pygame.mixer.music.stop* permet d'arrêter la lecture du fond sonore.
- *pygame.mixer.music.pause* permet de faire une pause dans la lecture.
- *pygame.mixer.music.set_volume* permet de régler le volume du fond sonore.
- *pygame.mixer.music.get_volume* permet de connaître le volume courant du fond sonore.

2.1.2 Le module pygame.mixer (effets sonores)

Pour les effets sonores, il faut préalablement créer (instancier) un objet de type *Sound* avant d'accéder à un ensemble de fonctions comparables à celles du sous-module *music*.

pygame.mixer.Sound permet de créer un nouvel objet de type *Sound*.

Une fois créé l'objet de type *Sound*, on peut utiliser une de ses fonctions. Entre autres :

- *pygame.mixer.Sound.play* permet de jouer le son.
- *pygame.mixer.Sound.stop* permet de stopper la diffusion du son.

2.2 Les fichiers son

Voici un court rappel sur les différents formats et extensions de fichiers sonores qui sont utilisables avec Pygame.

Deux formats sont particulièrement recommandés :

- le format WAV (*Waveform Audio File Format*)
- le format ouvert et libre OGG

Ils sont recommandés, car leur utilisation ne pose aucun problème, quelle que soit la plateforme. Il n'en va pas de même du format de compression audio MP3 par exemple, qui selon la documentation officielle peut induire des soucis d'utilisation sous certaines plateformes, en particulier avec la distribution Linux Debian (*« On some systems an unsupported format can crash the program, e.g. Debian Linux. Consider using OGG instead. »*).

Comment obtenir ou afficher des fichiers son pour un fond sonore ou pour des effets sonores ? On peut par exemple obtenir de la musique publiée sous licence libre sur Internet. On peut également créer ses propres fichiers son. Par exemple en les enregistrant grâce à l'enregistreur d'un smartphone ou en utilisant le logiciel libre d'enregistrement Audacity, qui permet aussi de faire du montage audio et d'exporter des sons dans le format de son choix, le format OGG en particulier.

2.3 La notion de channel (canal) dans Pygame

Pygame offre une notion supplémentaire dans la gestion du son. Il s'agit de la notion de channel (canal, en français). Un jeu possède plusieurs canaux son. Ainsi, on peut affecter tel son au canal numéro 1 et tel autre son au canal numéro 2. Il est ainsi possible de jouer simultanément des sons en activant leurs lectures sur des canaux différents comme le suggère l'exemple théorique suivant.

```
pygame.mixer.set_num_channels(2)

sound_1 = pygame.mixer.Sound("son1.ogg")
sound_2 = pygame.mixer.Sound("son2.ogg")

canal_1 = pygame.mixer.Channel(0)
canal_2 = pygame.mixer.Channel(1)

canal_1.play(sound_1)
canal_2.play(sound_2)
```

Remarque

Cette notion est largement abordée dans les sections Le module mixer et music au chapitre Les principaux modules Pygame.

3. Exemple d'utilisation du son avec Pygame

Le but de ce petit exemple est d'utiliser les deux aspects présentés précédemment. Ainsi nous allons créer une fenêtre de jeu à laquelle on associe un fond sonore (*music*) ainsi que quelques effets sonores déclenchés par l'appui de boutons du clavier.

- Le fond sonore est un sifflement diffusé en boucle.
- L'appui sur la touche o diffuse un cocorico.
- L'appui sur la touche c diffuse un bruit de corneille.
- L'appui sur la touche v diffuse une sonnette de vélo.

En appuyant sur la touche [Flèche en haut], on augmente le volume de chacun des quatre sons. En appuyant sur la touche [Flèche en bas], on diminue ce même volume.

On commence nécessairement par initialiser le module *mixer*.

```
pygame.mixer.init()
```

Puis on crée le fond sonore d'après un fichier son OGG qui inclut un sifflement mélodique d'une durée de quelques dizaines de secondes.

```
SIFFLEMENT = pygame.mixer.music.load("sifflement.ogg")
```

Ensuite, on peut jouer ce fond sonore avec la fonction *play* qui prend deux paramètres :

- Le premier permet d'indiquer combien de fois on désire boucler sur le son (ici 10 fois).
- Le second, une valeur décimale, indique en secondes le moment du morceau qui constitue le début de la diffusion.

```
pygame.mixer.music.play(10, 0.0)
```

Nous définissons maintenant les trois effets sonores.

```
COQ = pygame.mixer.Sound("coq.ogg")
CORNEILLE = pygame.mixer.Sound("corneille.ogg")
VELO = pygame.mixer.Sound("vélo.ogg")
```

Ils sont déclenchés par l'appui d'une touche du clavier dédiée.

```
elif event.key == pygame.K_o:
 COQ.play()
elif event.key == pygame.K_c:
 CORNEILLE.play()
elif event.key == pygame.K_v:
 VELO.play()
```

Enfin, on gère le volume en l'augmentant ou en le diminuant, en commençant par obtenir le volume courant auquel on ajoute ou on retranche 0.1. En effet, le volume s'exprime entre 0.0 et 1.0 ; par défaut, le volume est à 1.0.

```
elif event.key == pygame.K_DOWN:
 VOLUME = pygame.mixer.music.get_volume() - 0.1

 pygame.mixer.music.set_volume(VOLUME)
 COQ.set_volume(VOLUME)
 CORNEILLE.set_volume(VOLUME)
 VELO.set_volume(VOLUME)

elif event.key == pygame.K_UP:
 VOLUME = pygame.mixer.music.get_volume() + 0.1

 pygame.mixer.music.set_volume(VOLUME)
 COQ.set_volume(VOLUME)
 CORNEILLE.set_volume(VOLUME)
 VELO.set_volume(VOLUME)
```

Le code complet du programme est le suivant :

```
import pygame, sys

pygame.init()
pygame.mixer.init()

# COULEURS
COULEUR_BLANCHE = pygame.Color(255, 255, 255)

# FENETRE DE 400 SUR 400
ECRAN = pygame.display.set_mode((400,400))
ECRAN.fill(COULEUR_BLANCHE)
pygame.display.set_caption("Chapitre 5, du son")

SUITE = True

# FOND SONORE
SIFFLEMENT = pygame.mixer.music.load("sifflement.ogg")
pygame.mixer.music.play(1, 0.0)

# EFFETS SONORES
COQ = pygame.mixer.Sound("coq.ogg")
CORNEILLE = pygame.mixer.Sound("corneille.ogg")
VELO = pygame.mixer.Sound("vélo.ogg")

# BOUCLE DE JEU
while SUITE:

 for event in pygame.event.get():
   if event.type == pygame.QUIT:
     SUITE = False
   elif event.type == pygame.KEYDOWN:
     if event.key == pygame.K_ESCAPE:
       SUITE = False
     elif event.key == pygame.K_o:
       COQ.play()
     elif event.key == pygame.K_c:
       CORNEILLE.play()
```

```
        elif event.key == pygame.K_v:
            VELO.play()
        elif event.key == pygame.K_DOWN:
            VOLUME = pygame.mixer.music.get_volume() - 0.1
            pygame.mixer.music.set_volume(VOLUME)
            COQ.set_volume(VOLUME)
            CORNEILLE.set_volume(VOLUME)
            VELO.set_volume(VOLUME)
        elif event.key == pygame.K_UP:
            VOLUME = pygame.mixer.music.get_volume() + 0.1
            pygame.mixer.music.set_volume(VOLUME)
            COQ.set_volume(VOLUME)
            CORNEILLE.set_volume(VOLUME)
            VELO.set_volume(VOLUME)

    pygame.display.flip()
```