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 4
Maîtriser la librairie NumPy

1. Introduction à NumPy

NumPy est la librairie Python dédiée au calcul scientifique fournissant des fonctions très performantes de calcul, mais aussi des structures de données, tout aussi performantes.

En Data Science, il est essentiel d'avoir des structures adaptées pour stocker et manipuler de grandes quantités de données. C'est là qu'intervient NumPy, qui intègre une nouvelle structure de données en Python, les `ndarrays` (tableaux à N dimensions, en français, N représentant un chiffre), qui sont des tableaux multidimensionnels ou matrices. Il est important de noter que cette structure de données permet de stocker des données uniquement de même type (uniquement des nombres entiers par exemple).

Les `ndarrays` sont optimisés pour le stockage de données, mais aussi pour leur manipulation et plus encore pour les calculs, car ceux-ci peuvent être vectorisés. NumPy peut gérer de très gros tableaux et est très performante en temps de calcul sur ces tableaux : les `ndarrays` prennent en effet moins de place mémoire que d'autres objets Python, comme par exemple les listes. C'est pour cela que cette librairie a été développée et qu'elle est si utilisée : elle est très performante. C'est également pour cette raison que de nombreuses librairies ont été développées au-dessus de celle-ci, telle que Pandas, que nous verrons plus tard dans ce livre.

Les `ndarrays` de NumPy peuvent être unidimensionnels (aussi appelés 1D array, ce qu'on peut voir comme une liste), bidimensionnels (2D array) donc un tableau avec des lignes et des colonnes, ou encore des tableaux à plus de deux dimensions (3D array, 4D array, 5D array...), que nous ne verrons pas dans ce livre. Nous travaillerons exclusivement avec les `ndarrays` à deux dimensions dans ce chapitre, qui est la structure de données la plus utilisées en Data Science pour manipuler de grands jeux de données.

Lorsque vous souhaitez effectuer des opérations mathématiques ou logiques rapides sur un grand jeu de données, NumPy est votre allié. Pour pouvoir utiliser NumPy sous Python, il suffit de charger la librairie sous Jupyter, celle-ci étant déjà installée dans la distribution Anaconda.

Syntaxe

```
import numpy as np
```

Remarque

Il est courant d'utiliser l'alias "np" pour la librairie NumPy.

Remarque

Pour la suite de ce chapitre, n'hésitez pas à tester les codes que nous proposerons directement dans le notebook lié à ce chapitre et disponible en téléchargement.

2. Les tableaux NumPy

2.1 Créer un ndarray

2.1.1 Créer un ndarray à partir de listes

On peut créer un tableau NumPy à partir d'une liste en utilisant la fonction `array()` afin de convertir cette liste en tableau.

Syntaxe

```
import numpy as np
notre_tableau=np.array([élément1,élément2,élément3,élément4])
```

Exemple de code

```
import numpy as np
notre_tableau=np.array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,
18,19,20])
print(notre_tableau)
type(notre_tableau)
```

Résultat

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
numpy.ndarray
```

Ici, on importe la librairie NumPy, on crée un tableau à partir d'une liste, on affiche ce tableau avec `print()` et on affiche le type de l'objet `notre_tableau`. En résultat, on a notre premier array NumPy, qui est un tableau à une dimension ici, qu'on pourrait considérer comme une liste, mais qui est bien de type `ndarray`.

Pour créer un tableau bidimensionnel, il faut créer une liste contenant des listes, ce qui permet de représenter un tableau à deux entrées : les lignes et les colonnes. Pour cela, il suffit de considérer que la liste qu'on donne à la fonction `array()` est une liste de lignes et que les listes contenues dans cette liste représentent les colonnes. Ainsi, chaque ligne contient une liste qui correspond aux colonnes de cette ligne.

Syntaxe

```
mon_array_bidimensionnel=np.array([[élément ligne 1 colonne 1,
élément ligne 1 colonne 2,élément ligne 1 colonne 3],
[élément ligne 2 colonne 1, élément ligne 2 colonne 2,
élément ligne 2 colonne 3]])
```

Code

```
import numpy as np
mon_array_bidimensionnel=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(mon_array_bidimensionnel)
```

Résultat

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

On remarque tout de suite que, visuellement, cela ressemble à un tableau à deux dimensions, dont la première ligne contient trois colonnes avec les chiffres 1,2,3, la deuxième ligne (la deuxième liste de la liste principale) également trois colonnes avec les chiffres 4, 5, 6, etc.

Il s'agit d'un tableau d'entiers ici, et on ne pourrait pas changer une valeur par un type caractère par exemple, cela générerait une erreur.

Code

```
mon_array_bidimensionnel[0,0]="texte"
```

Résultat

```
ValueError: invalid literal for int() with base 10: 'texte'
```

Ici, nous disons à Python que nous souhaitons modifier la valeur de la première ligne à la première colonne par la chaîne de caractères "texte". Python nous retourne une erreur en nous disant que "texte" n'est pas de type int (entier), donc ça ne fonctionne pas. Comme nous le disions, un ndarray ne peut contenir qu'un seul type de données. Pour connaître le type des données contenues dans un ndarray, il suffit d'utiliser l'attribut dtype.

Syntaxe

```
mon_array.dtype
```

Regardons le type de données de notre tableau d'exemple.

Exemple de code

```
mon_array_bidimensionnel.dtype
```

Résultat

```
dtype('int32')
```

Ici, le type est int32, notre tableau ne peut donc contenir que des nombres entiers, d'où l'erreur précédente.

Remarque

Il existe deux types `int` : `int32` et `int64`. La différence entre `int32` et `int64`, c'est globalement la capacité de stockage du type. `int32` peut stocker des nombres entiers entre -2 147 483 648 et 2 147 483 647 et `int64` peut stocker des nombres entiers entre -9 223 372 036 854 775 808 et +9 223 372 036 854 775 808.

Il est possible de spécifier le type des données du tableau qu'on souhaite créer, avec l'option `dtype` dans la fonction `array()` de NumPy.

Code

```
mon_array_bidimensionnel=np.array([[1.5,2,3],[4,5.2,6],[7,8,9.1]],
dtype = float)
print(mon_array_bidimensionnel)
mon_array_bidimensionnel.dtype
```

Ici, on crée un tableau en spécifiant le type, `float`, puis on affiche le contenu de ce tableau. Enfin, on affiche le type des données contenues dans ce tableau.

Résultat

```
[[1.5 2.  3. ]
 [4.  5.2 6. ]
 [7.  8.  9.1]]
Out[9]:
dtype('float64')
```

Il s'agit bien d'un tableau contenant des données de type `float`, donc des nombres à virgule. Tous les entiers qu'on a donnés lors de la création du tableau (par exemple 2, 3, 4...) sont transformés en réels, `float`, et sont donc suivis de ".0".

2.1.2 Créer un ndarray grâce à des fonctions NumPy

Il arrive que vous sachiez d'avance la taille de votre `ndarray` mais pas encore son contenu, car vous souhaitez remplir ce tableau au fur et à mesure de votre code. NumPy fournit des fonctions permettant de créer et initialiser des `ndarrays` de taille connue et de contenu inconnu.

La première fonction est `np.zeros()`. Cette fonction permet de créer un `ndarray` dont l'ensemble des éléments correspondent à la valeur zéro.

Syntaxe

```
np.zeros((nombre de lignes, nombre de colonnes))
```

Il suffit de donner le nombre de lignes et de colonnes que vous souhaitez voir apparaître dans votre tableau.

Code

```
import numpy as np
mon_array_zeros=np.zeros((4,6))
print(mon_array_zeros)
```

Résultat

```
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]]
```

On obtient un tableau de type `float`, par défaut, rempli de zéros et de dimensions quatre lignes et six colonnes. Si on veut, on peut spécifier que le tableau est de type `int`.

Code

```
import numpy as np
mon_array_zeros=np.zeros((4,6), dtype=int)
print(mon_array_zeros)
```

Résultat

```
[[0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]]
```

Remarque

Il est aussi possible de créer un ndarray à une dimension contenant uniquement des zéros. Pour cela, plutôt que de donner un tuple contenant le nombre de lignes et de colonnes, il suffit de donner un seul nombre à la fonction `np.zeros` : `np.zeros(5)` créera un tableau à une dimension contenant 5 valeurs 0.

Il existe ensuite la fonction `np.ones()`. Cette fonction permet de créer un `ndarray` dont l'ensemble des éléments correspondent au chiffre 1.

Syntaxe

```
np.ones((nombre de lignes, nombre de colonnes))
```

Exemple de code

```
import numpy as np
mon_array_ones=np.ones((4,6), dtype=int)
print(mon_array_ones)
```

Résultat

```
[[1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]]
```

Enfin, il existe la fonction `np.emptys()`. Cette fonction permet de créer un `ndarray` dont les valeurs des éléments sont aléatoires.

Syntaxe

```
np.empty((nombre de lignes, nombre de colonnes))
```

Exemple de code

```
import numpy as np
mon_array_empty=np.empty((4,6), dtype=int)
print(mon_array_empty)
```

Résultat

```
[[-1805877096         611          64           0           0           0]
 [          0           0           0           0   929446194   912471142]
 [ 1667510839   909654373   909193776  1664104498  1630941233   929260599]
 [  959657314   878982705  1650615603   929457204  1697985846   842152292]]
```