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 :

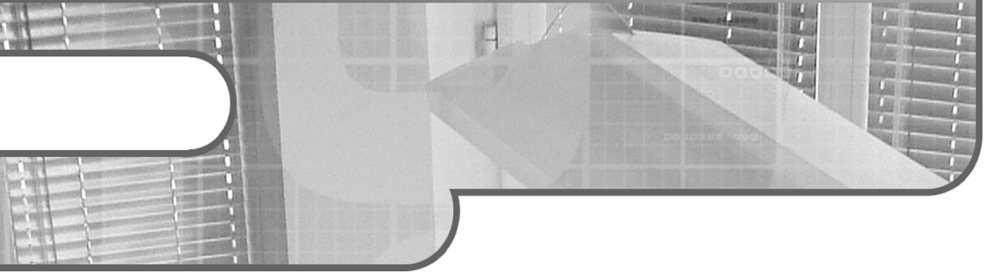
```
def demander_saisie_nombre(invite, minimum, maximum):
    invite += " entre " + str(minimum) + " et " +
              str(maximum) + " : "

    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 2.2
Programmation réseau

1. Écrire un serveur et un client

1.1 Utilisation d'une socket TCP

TCP est l'acronyme de *Transmission Control Protocol*, soit protocole de contrôle de transmission et a été développé en 1973 et documenté au sein de la RFC 793. Il est situé dans le modèle OSI au sein de la couche transport et est répandu de par sa fiabilité.

Il se caractérise par la manière de mettre en place la synchronisation entre client et serveur et par le découpage en segments des octets à transmettre, chaque segment étant parfaitement identifiable et disposant d'un système de contrôle d'intégrité qui fait que celui qui reçoit un paquet peut savoir que le paquet est corrompu et le redemander.

Le flux TCP utilise les sockets et c'est le module éponyme de Python qui va nous permettre de travailler avec TCP.

L'idée ici est de réaliser un mini serveur de données clé-valeur très basique et le plus simple possible de manière à voir comment créer un serveur puis un client, mais aussi comment manipuler les données qui sont échangées de l'un à l'autre.

Ce qu'il faut absolument comprendre est que les données qui sont transmises d'un serveur à un client ou d'un client à un serveur sont des octets, et rien d'autre. En Python 2, cela pouvait prêter à confusion, puisque le type **`str`** de Python 2 était assimilable à des octets. Le type Python 3 **`str`** représente une chaîne de caractères en unicode et c'est le type **`bytes`** qui permet de gérer des octets.

Il faut donc particulièrement être attentif à ce point, puisque la plupart des exemples présents sur le net, par ailleurs d'excellentes qualités, sont écrits pour Python 2 et génèrent une confusion sur le type de données réellement envoyées.

On va créer un serveur qui s'attend à recevoir un nombre et qui renvoie un code en fonction du fait que ce nombre soit premier ou non.

Voici le code ce cette fonction :

```
def isprime(n):
    '''check if integer n is a prime'''

    # negative numbers, 0 and 1 are not primes
    if n < 2:
        return False

    # 2 is the only even prime number
    if n == 2:
        return True

    # all other even numbers are not primes
    if not n & 1:
        return False

    # range starts with 3 and only needs to go up the squareroot
    # of n for all odd numbers
    for x in range(3, int(n**0.5)+1, 2):
        if n % x == 0:
            return False
    return True
```

Voici donc le code du serveur (disponible en téléchargement) :

```
#!/usr/bin/python3

import socket

params = ('127.0.0.1', 8809)
BUFFER_SIZE = 1024 # default

donnees = {}

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)#Internet, TCP
s.bind(params)
s.listen(1)
```

Le serveur est lancé et il écoute sur le port 8809. la méthode accept attend qu'une connexion se présente et dès qu'elle arrive, elle renvoie les éléments nécessaires pour que la connexion puisse être traitée. Ce qui est réalisé dans une boucle sans fin. Le principe est que tant que l'échange client serveur dure, il est traité.

```
conn, addr = s.accept()
print('Connexion acceptée: %s' % str(addr))

while True:
        data = conn.recv(BUFFER_SIZE)
        if not data:
            break
```

```
        print('Donnée reçue: %s' % data)
        try : number = int(data)
        except:
            response = b'E'
            phrase = "'%s' n'est pas un entier" % data
        else:
            if isprime(number):
                phrase = "'%s' est un nombre premier" % number
                response = b'T'
            else:
                phrase = "'%s' n'est pas est un nombre premier" % number
                response = b'F'
        finally:
            print(response)
            conn.send(response)
conn.close()
```

La partie complexe n'est pas tant la gestion de la connexion et des transferts réseau que le traitement des données. Ce qui compte est de savoir quoi faire des données et comment mettre en place un dialogue entre client et serveur, sachant que les seuls éléments échangeables sont des octets.

Dans cet exemple, on a choisi la convention suivante : si on ne comprend pas la donnée transmise par le client, on renvoie un code E (pour erreur). Si le nombre est premier, on renvoie un code T (pour True), sinon on renvoie un code F (pour False).

Les choix effectués dans cet exemple sont assez basiques et limités : on cherche à transmettre le moins de données possible, mais serveur et client doivent s'entendre pour se comprendre entre eux : le client devra connaître l'ensemble des codes susceptibles d'être envoyés par le serveur et les gérer.

Cela montre les problématiques typiques qui se posent lorsque l'on cherche à manipuler des données en utilisant des couches de bas niveau.

Voici le code du client, qu'il faut lire en faisant le lien avec celui du serveur :

```
#!/usr/bin/python3

import socket

params = ('127.0.0.1', 8809)
BUFFER_SIZE = 1024 # default

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(params)
```

Le parti a été pris d'envoyer une succession de messages de manière à couvrir tous les cas possibles de traitement côté serveur.

```
chiffres = [4j, 4, 5, -5, 17, 29, 2**50, 2**50-1]
```

En face de chaque message, qui est en réalité une sorte d'instruction donnée au serveur, est mis en commentaire le comportement attendu.
Voici le code qui effectue les requêtes vers le serveur et qui traite les réponses :

```
for chiffre in chiffres:
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(params)
    print("Envoi d'un message %s" % chiffre)
    s.send(('%s\n' % chiffre).encode('utf8'))
    data = s.recv(BUFFER_SIZE)
    if len(data) == 0:
        print("\tPas de réponses")
    elif data == b'E\n':
        print("\tUne erreur est détectée par le serveur")
    elif data == b'T\n':
        print("\tLe nombre est bien un nombre premier")
    elif data == b'F\n':
        print("\tLe nombre n'est pas un nombre premier")
    else:
        print("\tLa donnée reçue n'est pas comprise : %s" % data)
    s.close()
```

TODO : tant que l'on est dans la boucle, on communique avec le serveur et donc le serveur reste dans la boucle infinie de communication. Le fait de fermer la connexion côté client envoie une trame vide qui va permettre de passer dans le break côté serveur (**`if not data: break`**) et ainsi libérer le serveur.
S'il y avait une boucle autour de la méthode **`accept`**, le serveur continuerait à tourner en attendant de nouvelles données qu'une nouvelle connexion lui parvienne et tournerait ainsi à l'infini, mais en l'occurrence, il n'y a pas cette boucle et il se stoppe donc.
Cet exemple ne permet donc que de parler à un seul client, puis d'arrêter le serveur. Il existe, dans la pratique, des cas d'utilisation tels que celui-ci.
Il existe, par exemple, un excellent outil nommé woof qui s'installe via le gestionnaire de paquet et se lance en ligne de commande :

```
$ sudo aptitude install woof
```

```
$ python path/to/woof.py path/to/filename
```

Un serveur est lancé et propose une URL qu'il suffit de donner à quelqu'un par IRC ou un autre moyen pour que cette personne, en cliquant dessus, télécharge le fichier. À la fin du téléchargement, le serveur se ferme. On peut donc transmettre un document par le réseau rapidement.

1.2 Utilisation d'une socket UDP

Comme TCP, UDP appartient à la couche transport du modèle OSI, mais contrairement à lui, il ne réalise pas de connexion préliminaire, de synchronisation et ne garantit pas la bonne réception des données. Par contre, il intègre un système qui assure l'intégrité de la donnée transmise, comme TCP.

UDP est moins fiable que TCP, car il peut perdre des paquets, mais il est beaucoup plus rapide car il nécessite beaucoup moins d'allers-retours. Il est donc préférable pour des technologies pour lesquelles la fiabilité n'est pas le principal besoin par rapport à la rapidité, comme la voix sur IP, le streaming ou les jeux en réseau.

Ceci est vrai dans la mesure où les applications concernées sont capables soit de se passer d'une petite partie des données qui auraient été perdues, soit de les reconstruire (à partir du moment où cela est plus rapide que les redemander), soit de les substituer.

D'autre part, TCP est utilisé pour établir un dialogue entre un client et un serveur alors qu'UDP est utilisé pour envoyer des données du client vers le serveur sans que le client attende un retour.

Là encore, on va utiliser les sockets, mais comme les protocoles TCP et UDP diffèrent dans leur fonctionnement, cette différence est visible dans la manière d'écrire un serveur et un client.

Ainsi, le serveur n'attend pas de synchroniser une connexion avec un client et le client ne se connecte pas au serveur avant de lui envoyer des données.

En dehors de cela, la manière de traiter les données est identique à ce que l'on a vu pour TCP. Des exemples plus simples que ceux montrés dans cet ouvrage permettent de mettre en évidence les différences entre les protocoles TCP (http://wiki.python.org/moin/TcpCommunication) et UDP (http://wiki.python.org/moin/UdpCommunication). Dans les deux cas, les serveurs affichent la donnée reçue et pour TCP, la donnée est également renvoyée au client.

L'exemple suivant est dans la lignée de celui pour TCP, le client donnant des informations au serveur. Les lignes inutiles sont laissées en commentaire afin de mettre en évidence les différences avec TCP.

```
#!/usr/bin/python3

import socket

params = ('127.0.0.1', 8808)
BUFFER_SIZE = 1024 # default

donnees = {}

s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM) #SOCK_STREAM
s.bind(params)
# Les lignes suivantes n'ont pas de sens en UDP:
#s.listen(1)
#conn, addr = s.accept()
#print('Connexion acceptée: %s' % str(addr))
```

On voit clairement dans la première partie qu'il n'est pas nécessaire de se mettre à l'écoute et d'initier une connexion.
Dans la partie qui suit, la différence de sémantique entre **recv** et **recvfrom** est révélatrice du fait que la méthode **recvfrom** reçoit à la fois la donnée et l'adresse du client.

```
while True:
        #data = conn.recv(BUFFER_SIZE)
        data, addr = s.recvfrom(BUFFER_SIZE)
        print('Connexion reçue: %s' % str(addr))
        print('Donnée reçue: %s' % data)
        if not data:
            break
        print('Donnée reçue: %s' % data)
        try:
            number = int(data)
        except:
            response = b'E'
            phrase = "'%s' n'est pas un entier" % data
        else:
            if isprime(number):
                phrase = "'%s' est un nombre premier" % number
                response = b'T'
            else:
                phrase = "'%s' n'est pas est un nombre premier" % number
                response = b'F'
        finally:
            print(response)
#conn.close()
```

À noter que si pour TCP il faut une boucle pour gérer chaque connexion et une autre pour gérer les échanges au sein de la connexion, en UDP une seule boucle suffit. Pour que l'exemple puisse se terminer proprement, il a été rajouté un cas d'utilisation particulier permettant de sortir de la boucle.
Voici la partie cliente correspondant :

```
#!/usr/bin/python3

import socket
params = ('127.0.0.1', 8808)
BUFFER_SIZE = 1024 # default

s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
#s.connect(params)
```

Là encore, pas besoin de connexion. C'est au moment où l'on envoie les données que l'on donne les informations sur le serveur à atteindre.

```
chiffres = [4j, 4, 5, -5, 17, 29, 2**50, 2**50-1]
```