Les fonctions

NSI – Première

Définir, spécifier, tester et réutiliser des fonctions

Table des matières

Introduction

Depuis le début de l'année, tous les programmes que vous avez écrits sont composés d'une unique suite d'instructions, exécutées les unes après les autres, du début à la fin. Cette façon de faire fonctionne pour de petits programmes, mais elle montre vite ses limites dès que :

— un même calcul, ou un même bloc de code, doit être répété à plusieurs endroits du programme (au prix de copier-coller, source d'erreurs et de code difficile à corriger) ;
— le programme devient long, et il devient difficile de s'y retrouver, de le tester, ou de le faire évoluer ;
— on souhaite réutiliser, dans un autre programme, un morceau de code déjà écrit et déjà testé.

La notion de **fonction** répond à ces trois problèmes : elle permet de découper un programme en blocs indépendants, nommés, réutilisables et testables séparément. C'est l'un des outils les plus importants de tout langage de programmation.

Dans cette séquence, nous allons apprendre à définir nos propres fonctions en Python, à les documenter précisément (on parle de **spécification**), à vérifier qu'elles fonctionnent correctement à l'aide de **jeux de tests**, puis à les regrouper dans nos propres **bibliothèques** de code réutilisable.

Remarque : Activité d'introduction

L'activité d'introduction de cette séquence reprend le code du jeu du pendu, écrit lors du TP de synthèse du chapitre sur les bases de python, et vous demande de le réécrire en utilisant des fonctions. C'est l'occasion idéale de voir concrètement ce que les fonctions apportent : un code plus court, plus lisible, et plus facile à corriger.

1 Qu'est-ce qu'une fonction ?

1.1 Une notion déjà connue en mathématiques

En mathématiques, une fonction f associe, à un nombre x, un unique nombre noté $f(x)$. Par exemple, $f(x) = x^2$ associe à 3 la valeur $f(3) = 9$.

En informatique, la notion de fonction reprend cette idée : une fonction associe, à une ou plusieurs **valeurs d'entrée**, un **résultat**. Mais une fonction informatique est plus générale qu'une fonction mathématique : elle peut prendre plusieurs valeurs en entrée (ou aucune), et exécuter n'importe quelle suite d'instructions (affichages, boucles, tests, etc.), pas seulement un calcul.

Définition 1.1: Fonction (en informatique)

Une **fonction** est un sous-programme, identifié par un **nom**, qui peut :
- recevoir des valeurs en entrée, appelées **paramètres** ;
- exécuter une suite d'instructions ;
- **renvoyer** une valeur en sortie, à l'aide du mot-clé `return`.

Exécuter le code d'une fonction depuis un autre endroit du programme s'appelle **appeler** la fonction.

Syntaxe : Définir et appeler une fonction en Python

```
1 def nom_fonction(parametre_1, parametre_2):
2     # instructions
3     resultat = ...
4     return resultat
5
6 # appel de la fonction :
7 nom_fonction(valeur_1, valeur_2)
```

Le mot-clé `def` (pour *define*) introduit la définition d'une fonction. Le bloc d'instructions qui suit est indenté, comme pour une boucle ou un `if`.

Exemple 1.2: Une première fonction

```
1 def carre(x):
2     return x * x
3
4 print(carre(5))       # affiche 25
5 y = carre(3) + 1
6 print(y)               # affiche 10
```

Chaque appel `carre(5)` est **remplacé** par la valeur renvoyée par la fonction : $5 \times 5 = 25$.

Remarque : Paramètre et argument

On distingue parfois deux mots proches, mais différents :
- le **paramètre** est le nom utilisé *dans la définition* de la fonction (ici, `x`) ;
- l'**argument** est la valeur effectivement transmise *lors de l'appel* (ici, `5`, puis `3`).

Dans la pratique, les deux mots sont souvent employés l'un pour l'autre.

Exercice 1.3: Tracer l'exécution d'une fonction

On considère la fonction suivante :

```
1 def mystere(a, b):
```

```
    somme = a + b
    produit = a * b
    return somme - produit

print(mystere(3, 4))
print(mystere(-2, 5))
```

Donner la valeur renvoyée par chacun des deux appels, en détaillant le calcul.

Correction

Pour `mystere(3, 4)` : `somme = 3 + 4 = 7` et `produit = 3 * 4 = 12`, donc la fonction renvoie $7 - 12 = \mathbf{-5}$.
Pour `mystere(-2, 5)` : `somme = -2 + 5 = 3` et `produit = -2 * 5 = -10`, donc la fonction renvoie $3 - (-10) = \mathbf{13}$.

Exercice 1.4: Écrire des fonctions simples

Écrire une fonction `double(n)` qui renvoie le double de l'entier `n`, puis une fonction `est_pair(n)` qui renvoie `True` si `n` est pair, et `False` sinon.

Correction

```
def double(n):
    return 2 * n

def est_pair(n):
    return n % 2 == 0
```

Pour `est_pair`, on peut directement renvoyer le résultat du test `n % 2 == 0` (qui vaut déjà `True` ou `False`), sans passer par un `if`.

1.2 Le piège du *print* à la place du *return*

Remarque : Afficher n'est pas renvoyer

C'est l'erreur la plus fréquente chez les débutants : confondre **afficher** un résultat à l'écran (`print`) et le **renvoyer** pour qu'il puisse être réutilisé (`return`).

Exemple 1.5: Une fonction qui ne renvoie rien

```
def carre_faux(x):
    print(x * x)      # affiche le résultat, mais ne le renvoie pas

resultat = carre_faux(5)    # affiche 25 à l'écran
print(resultat)               # affiche None !
```

`carre_faux(5)` affiche bien `25` à l'écran, mais la fonction ne se termine par aucun `return` : elle renvoie donc la valeur spéciale `None`, qui signifie « aucune valeur ». La variable `resultat` vaut `None`, et non `25` : elle est inutilisable dans un calcul.

Définition 1.6: La valeur *None*

`None` est une valeur spéciale de Python qui représente l'**absence de valeur**. Une fonction qui ne contient aucun `return` (ou un `return` sans expression derrière) renvoie automatiquement `None`.

Méthode 1.7: Choisir entre *print* et *return*

- On utilise `print` uniquement pour **afficher** une information à l'écran, à destination de l'utilisateur.
- On utilise `return` pour **renvoyer** un résultat, afin qu'il puisse être réutilisé ailleurs dans le programme (stocké dans une variable, réinjecté dans un calcul, dans un test, etc.).
- Une fonction destinée à **calculer** un résultat doit donc, presque toujours, se terminer par un `return`, et non par un `print`.

Exercice 1.8: Corriger un piège classique

Le code suivant doit calculer la somme des trois nombres `4`, `7` et `2`, mais il provoque une erreur :

```
1 def somme(a, b):
2     print(a + b)
3
4 total = somme(somme(4, 7), 2)
5 print(total)
```

1. Expliquer précisément pourquoi ce code ne fonctionne pas comme prévu.
2. Corriger la fonction `somme` pour que le programme affiche bien `13`.

Correction

1. `somme(4, 7)` affiche `11` à l'écran, mais renvoie `None` (aucun `return`). L'appel `somme(None, 2)` tente alors de calculer `None + 2`, ce qui provoque une erreur (`TypeError`) : on ne peut pas additionner `None` avec un entier.
2. Il suffit de remplacer `print` par `return` :

```
1 def somme(a, b):
2     return a + b
3
4 total = somme(somme(4, 7), 2)
5 print(total)   # affiche 13
```

2 Portée des variables

Lorsqu'on définit plusieurs fonctions, une question se pose naturellement : une variable créée dans une fonction est-elle visible en dehors de celle-ci ? Et inversement ?

Définition 2.1: Variable locale, variable globale

— Une variable créée **à l'intérieur** d'une fonction (y compris ses paramètres) est une variable **locale** : elle n'existe que pendant l'exécution de cette fonction, et n'est pas accessible depuis l'extérieur.

— Une variable créée **en dehors** de toute fonction, au niveau principal du programme,

est une variable **globale**.
On appelle **portée** d'une variable la partie du programme dans laquelle elle est accessible.

Exemple 2.2: Une variable locale n'existe pas à l'extérieur

```
1 def f():
2     x = 10
3     print(x)     # affiche 10, x existe ici
4
5 f()
6 print(x)          # erreur : NameError, x n'existe pas ici
```

La variable `x` est créée **à l'intérieur** de `f` : elle est locale à `f`, et disparaît dès que l'exécution de `f` se termine.

Remarque : Deux fonctions peuvent réutiliser les mêmes noms

Puisque les variables locales de chaque fonction sont indépendantes, deux fonctions différentes peuvent tout à fait utiliser un paramètre ou une variable portant le même nom (par exemple `x`) sans jamais se perturber l'une l'autre : chacune a sa propre version de `x`, invisible depuis l'autre fonction.

Exemple 2.3: Lecture d'une variable globale

```
1 compteur = 0
2
3 def afficher_compteur():
4     print(compteur)    # lecture d'une variable globale : autorisée
5
6 afficher_compteur()     # affiche 0
```

Une fonction peut **lire** une variable globale sans problème.

Remarque : Éviter de modifier une variable globale

Modifier une variable globale depuis l'intérieur d'une fonction est possible en Python (à l'aide du mot-clé `global`), mais c'est une pratique à éviter : le comportement de la fonction dépend alors de variables extérieures invisibles dans sa définition, ce qui rend le programme difficile à comprendre et à corriger. On préfère toujours faire transiter les données par les **paramètres** (en entrée) et le `return` (en sortie).

Exercice 2.4: Portée des variables

On considère le code suivant :

```
1 x = 5
2
3 def double_x():
4     x = 2 * x
5     return x
6
7 print(double_x())
8 print(x)
```

1. Que va afficher ce programme ? Justifier.
2. Proposer une version corrigée de `double_x` qui renvoie bien le double de la valeur

qu'on lui donne, en utilisant un **paramètre**.

Correction

1. Ce programme provoque une erreur (`UnboundLocalError`). En effet, la ligne `x = 2 * x` crée `x` comme variable **locale** à `double_x` (car on lui affecte une valeur à l'intérieur de la fonction) ; mais au moment de calculer `2 * x`, Python cherche la valeur de cette variable locale `x`, qui n'a pas encore été définie. La variable globale `x = 5` n'est donc pas utilisée ici.
2. On utilise un paramètre, ce qui évite toute ambiguïté :

```
1  x = 5
2
3  def double_x(n):
4      return 2 * n
5
6  print(double_x(x))    # affiche 10
7  print(x)                # affiche toujours 5
```

3 Décomposer un programme en fonctions

L'un des principaux intérêts des fonctions est de permettre de **découper** un programme en blocs indépendants, plutôt que d'écrire une longue suite d'instructions.

Exemple 3.1: Du code dupliqué...

```
1  longueur1, largeur1 = 5, 3
2  aire1 = longueur1 * largeur1
3  print("Aire du rectangle 1 :", aire1)
4
5  longueur2, largeur2 = 8, 2
6  aire2 = longueur2 * largeur2
7  print("Aire du rectangle 2 :", aire2)
8
9  longueur3, largeur3 = 4, 4
10 aire3 = longueur3 * largeur3
11 print("Aire du rectangle 3 :", aire3)
```

Le même calcul (`longueur * largeur`) et le même affichage sont répétés trois fois.

Exemple 3.2: ... à une fonction réutilisable

```
1  def aire_rectangle(longueur, largeur):
2      return longueur * largeur
3
4  print("Aire du rectangle 1 :", aire_rectangle(5, 3))
5  print("Aire du rectangle 2 :", aire_rectangle(8, 2))
6  print("Aire du rectangle 3 :", aire_rectangle(4, 4))
```

Le calcul n'est écrit **qu'une seule fois**, dans la fonction. S'il contenait une erreur, il suffirait de la corriger à un seul endroit.

Méthode 3.3: Repérer un bloc de code à transformer en fonction

- Le même bloc de code (identique, ou presque) apparaît **plusieurs fois** dans le programme ⇒ on peut le remplacer par une fonction, appelée à chaque endroit concerné.
- Un bloc de code réalise une tâche que l'on sait **nommer** clairement (« calculer l'aire », « vérifier si le mot est trouvé », etc.) ⇒ on peut l'isoler dans une fonction portant ce nom, même s'il n'apparaît qu'une seule fois.

Propriété 3.4: Intérêts de la décomposition en fonctions

Découper un programme en fonctions permet :
- de **réduire la duplication** de code (moins de code ⇒ moins d'erreurs) ;
- d'obtenir un code plus **lisible** : chaque fonction porte un nom qui explique ce qu'elle fait ;
- de **tester** chaque fonction séparément, indépendamment du reste du programme ;
- de **réutiliser** une fonction déjà écrite dans un autre programme, sans avoir à la réécrire.

Remarque : Vers l'activité d'introduction

C'est exactement cette démarche que vous allez mettre en pratique dans l'activité d'introduction de cette séquence : vous reprendrez le code du jeu du pendu (TP de synthèse du chapitre précédent), qui mélange affichage du mot à trous, gestion des essais et détection de la victoire au même niveau, et vous en extrairez des fonctions bien identifiées.

Exercice 3.5: Extraire une fonction

Réécrire le code suivant en définissant une fonction `perimetre_carre(cote)`, puis en l'utilisant pour afficher les trois périmètres :

```
1 cote1 = 6
2 print("Périmètre 1 :", 4 * cote1)
3
4 cote2 = 10
5 print("Périmètre 2 :", 4 * cote2)
6
7 cote3 = 3
8 print("Périmètre 3 :", 4 * cote3)
```

Correction

```
1 def perimetre_carre(cote):
2     return 4 * cote
3
4 print("Périmètre 1 :", perimetre_carre(6))
5 print("Périmètre 2 :", perimetre_carre(10))
6 print("Périmètre 3 :", perimetre_carre(3))
```

4 Spécifier une fonction

Avant même d'écrire le code d'une fonction, il est indispensable de savoir précisément ce qu'elle doit faire : quelles données elle attend, sous quelle forme, et ce qu'elle doit renvoyer. C'est ce qu'on appelle **spécifier** une fonction.

Définition 4.1: Prototype (ou signature) d'une fonction

Le **prototype** (ou **signature**) d'une fonction est la description de :
- son **nom** ;
- ses **paramètres**, avec le type de valeur attendu pour chacun ;
- le **type** de la valeur qu'elle renvoie.

Notation 4.2: Indication de types en Python

Python permet d'indiquer les types directement dans l'en-tête de la fonction, à l'aide de `:` pour un paramètre et de `->` pour la valeur renvoyée. Ces indications sont uniquement informatives : Python ne les vérifie pas à l'exécution.

```
1 def carre(x: int) -> int:
2     return x * x
```

Le prototype de `carre` se lit ainsi : « `carre` prend un entier `x` et renvoie un entier ».

Définition 4.3: Précondition, postcondition

- Une **précondition** est une condition que doivent vérifier les arguments pour que la fonction fonctionne correctement (par exemple : « `n` doit être un entier positif »).
- Une **postcondition** est une propriété garantie sur la valeur renvoyée, **à condition** que les préconditions aient été respectées.

Syntaxe : Documenter une fonction avec une *docstring*

On rédige la spécification d'une fonction directement sous sa ligne `def`, entre triples guillemets `""" ... """` : c'est ce qu'on appelle une **docstring**.

```
1  def racine_entiere(n):
2      """
3      Renvoie la partie entière de la racine carrée de n.
4      Précondition : n est un entier positif ou nul.
5      Postcondition : le résultat r vérifie r*r <= n < (r+1)*(r+1).
6      """
7      r = 0
8      while (r + 1) * (r + 1) <= n:
9          r = r + 1
10     return r
```

Remarque : Vérifier une précondition avec *assert*

On peut faire vérifier une précondition **par Python lui-même**, à l'aide du mot-clé `assert` : si la condition est fausse, le programme s'arrête immédiatement avec un message d'erreur explicite, plutôt que de produire un résultat incorrect silencieusement.

```
1 def racine_entiere(n):
2     assert n >= 0, "n doit être positif ou nul"
3     r = 0
4     while (r + 1) * (r + 1) <= n:
5         r = r + 1
6     return r
```

Exercice 4.4: Rédiger la spécification d'une fonction

On souhaite écrire une fonction `moyenne(a, b, c)` qui renvoie la moyenne arithmétique de trois notes, chacune comprise entre 0 et 20.

1. Donner le prototype de cette fonction (nom, paramètres et types, type de la valeur renvoyée).
2. Donner une précondition et une postcondition pertinentes.
3. Écrire le code complet de la fonction, avec sa docstring.

Correction

1. Prototype : `moyenne(a: float, b: float, c: float) -> float`.
2. Précondition : `a`, `b` et `c` sont des nombres compris entre 0 et 20. Postcondition : la valeur renvoyée est elle-même comprise entre 0 et 20.

```
13. def moyenne(a: float, b: float, c: float) -> float:
2       """
3       Renvoie la moyenne arithmétique de trois notes.
4       Précondition : a, b et c sont compris entre 0 et 20.
5       Postcondition : le résultat est compris entre 0 et 20.
6       """
7       assert 0 <= a <= 20 and 0 <= b <= 20 and 0 <= c <= 20
8       return (a + b + c) / 3
```

5 Tester une fonction

Écrire une fonction ne suffit pas : encore faut-il vérifier qu'elle fait bien ce qui est attendu, y compris dans des cas particuliers.

Définition 5.1: Jeu de tests

Un **jeu de tests** pour une fonction est un ensemble de cas (une entrée, associée à la sortie attendue) permettant de vérifier que la fonction se comporte correctement, y compris sur des **cas particuliers** : valeurs limites, valeur nulle, valeurs négatives, plus petite ou plus grande valeur possible, etc.

Syntaxe : Écrire un test avec *assert*

```
1 assert expression_a_tester == valeur_attendue
```

Si l'expression est vraie, rien ne se passe : le test est réussi. Si elle est fausse, Python lève une erreur (`AssertionError`) et interrompt le programme : le test a détecté un bug.

Exemple 5.2: Jeu de tests pour *carre*

```
1 def carre(x):
2     return x * x
3
4 assert carre(5) == 25
5 assert carre(0) == 0
6 assert carre(-3) == 9
7 print("Tous les tests sont passés !")
```

On teste ici une valeur « normale » (`5`), un cas particulier (`0`) et une valeur négative (`-3`),

pour laquelle le résultat reste positif.

Méthode 5.3: Choisir de bons cas de test

- Tester au moins un cas « normal », représentatif d'une utilisation courante.
- Tester les **cas limites** : zéro, valeur négative si elle est autorisée, plus petite et plus grande valeur possible.
- Si la fonction manipule des chaînes de caractères ou des séquences, penser au cas de la chaîne ou de la séquence **vide**.

Exercice 5.4: Écrire un jeu de tests

On rappelle la fonction `est_pair(n)` de la section 1, qui renvoie `True` si `n` est pair, et `False` sinon. Écrire, à l'aide d'`assert`, un jeu de tests d'au moins quatre cas pour cette fonction, en incluant des cas particuliers.

Correction

```
1 def est_pair(n):
2     return n % 2 == 0
3
4 assert est_pair(4) == True
5 assert est_pair(7) == False
6 assert est_pair(0) == True      # cas particulier : zéro est pair
7 assert est_pair(-2) == True     # cas particulier : un négatif pair
8 print("Tous les tests sont passés !")
```

Remarque : Vers l'activité d'introduction

Dans l'activité d'introduction, vous écrirez et exécuterez de tels jeux de tests pour vérifier que votre version « avec fonctions » du jeu du pendu produit exactement les mêmes résultats que la version originale.

6 Bibliothèques et modules

Définition 6.1: Bibliothèque (module)

Une **bibliothèque**, ou **module**, est un ensemble de fonctions déjà écrites et regroupées dans un fichier, que l'on peut **réutiliser** dans ses propres programmes sans avoir à les réécrire. Python est livré avec de nombreuses bibliothèques standards (`math`, `random`, etc.), et il en existe des milliers d'autres, écrites par la communauté.

Syntaxe : Importer une bibliothèque

```
1 import math
2 print(math.sqrt(16))      # 4.0 -- on accède à sqrt via math.
3
4 from math import sqrt, pi
5 print(sqrt(16))             # 4.0 -- plus besoin du préfixe math.
6
7 import random as rd
8 print(rd.randint(1, 6))    # un entier tiré au hasard entre 1 et 6
```

Les trois syntaxes ci-dessus sont possibles : `import module` (on préfixe chaque fonction par `module.`), `from module import fonction` (on importe une fonction précise, utilisable directement), et `import module as alias` (on renomme le module, souvent pour l'abréger).

Remarque : Utiliser la documentation d'une bibliothèque

On n'a jamais besoin de connaître **toutes** les fonctions d'une bibliothèque : il faut savoir consulter sa **documentation** pour trouver la fonction dont on a besoin, et connaître précisément ses paramètres et sa valeur de retour. Deux moyens simples d'y accéder :

— la documentation officielle de Python, sur `docs.python.org` ;
— la fonction `help(nom_fonction)`, directement dans l'interpréteur Python (par exemple `help(math.sqrt)`).

Définition 6.2: Créer sa propre bibliothèque

Un simple fichier `.py` contenant des définitions de fonctions peut lui-même servir de bibliothèque. Si un fichier nommé `conversions.py` se trouve dans le même dossier qu'un autre programme, on peut y accéder depuis ce programme avec `import conversions`, puis appeler par exemple `conversions.decimal_vers_binaire(58)`.

Exemple 6.3: Une mini-bibliothèque personnelle

Fichier `geometrie.py` :

```
1  def aire_rectangle(longueur, largeur):
2      """Renvoie l'aire d'un rectangle de côtés longueur et largeur."""
3      return longueur * largeur
4
5  def perimetre_rectangle(longueur, largeur):
6      """Renvoie le périmètre d'un rectangle de côtés longueur et largeur."""
7      return 2 * (longueur + largeur)
```

Dans un autre fichier, situé dans le même dossier :

```
1  import geometrie
2
3  print(geometrie.aire_rectangle(5, 3))        # 15
4  print(geometrie.perimetre_rectangle(5, 3))    # 16
```

Remarque : Vers le TP final

C'est exactement ce que vous ferez dans le TP final de cette séquence : vous construirez votre propre bibliothèque de fonctions de conversion entre les bases 10, 2 et 16 (une fonction paramétrée par la base visée), que vous pourrez ensuite importer et réutiliser dans n'importe quel autre programme.

Exercice 6.4: Utiliser une bibliothèque

1. Écrire les instructions permettant d'importer uniquement la fonction `randint` de la bibliothèque `random`, puis de l'utiliser pour afficher un entier aléatoire entre 1 et 100.
2. Le code suivant provoque une erreur. L'expliquer et le corriger.

```
1  import math
2  print(sqrt(25))
```

Correction

11.
```python
from random import randint
print(randint(1, 100))
```

2. Avec `import math`, la fonction `sqrt` n'est **pas** importée directement dans l'espace de noms principal : elle reste accessible uniquement via `math.sqrt`. Il faut donc écrire `print(math.sqrt(25))`, ou bien importer différemment avec `from math import sqrt`.

Bilan

Dans cette séquence, nous avons appris à :
- **définir** une fonction avec `def`, la faire communiquer avec le reste du programme grâce à ses **paramètres** et à son **return** ;
- distinguer une variable **locale** d'une variable **globale** ;
- **décomposer** un programme en fonctions, pour un code plus court, plus lisible et plus facile à corriger ;
- **spécifier** une fonction : prototype, préconditions, postconditions, docstring ;
- **tester** une fonction à l'aide d'un jeu de tests et d'`assert` ;
- utiliser une **bibliothèque** existante, et en créer une soi-même.