Fonctions en Python

Activité – Première NSI

① Pourquoi définir une fonction ?

Exercice 1 *Du code répété au code factorisé*

On souhaite calculer le périmètre de plusieurs cercles de rayons 3, 7 et 12 centimètres. Voici une première version, sans fonction :

```
from math import pi

p1 = 2 * pi * 3
print(f"Périmètre du cercle de rayon 3 : {p1}")

p2 = 2 * pi * 7
print(f"Périmètre du cercle de rayon 7 : {p2}")

p3 = 2 * pi * 12
print(f"Périmètre du cercle de rayon 12 : {p3}")
```

1. Combien de fois la formule $2 \times \pi \times r$ est-elle écrite ?
2. Que faudrait-il modifier si l'on apprenait que le bon facteur n'était pas 2π mais $2{,}01\pi$?
3. Réécrire ce programme en définissant une fonction `perimetre(rayon)` qui renvoie le périmètre d'un cercle de rayon donné. Le programme principal se contentera de trois appels à cette fonction.

Remarque 1. Une **fonction** regroupe un bloc d'instructions sous un nom. On évite ainsi de répéter les mêmes calculs et l'on rend le code plus lisible, plus maintenable et plus facile à tester.

② Définir et appeler une fonction

Définition 1 **(Définition d'une fonction)** En Python, une fonction se définit par le mot-clé `def` :

```
def nom_fonction(parametre_1, parametre_2, ...):
    """Documentation (optionnelle)."""
    # corps de la fonction
    return resultat
```

L'**appel** de la fonction utilise des parenthèses contenant les arguments : `nom_fonction(valeur_1, valeur_2, ...)`.

Exercice 2 *Première fonction*

On donne le code suivant :

```python
def double(x):
    return 2 * x

a = double(5)
b = double(a)
c = double(b + 1)
```

1. Que vaut `a` ? Justifier en indiquant la valeur passée en paramètre et la valeur renvoyée.
2. Que vaut `b` ?
3. Que vaut `c` ? Détailler le calcul effectué *avant* l'appel, puis le calcul effectué *dans* l'appel.
4. Si l'on écrit `print(double(double(3)))`, qu'affiche le programme ?

Exercice 3 *Compléter une fonction*

Compléter chacune des fonctions suivantes pour qu'elles fassent ce que dit leur nom.

```python
def aire_rectangle(longueur, largeur):
    """Renvoie l'aire d'un rectangle."""
    return ............

def moyenne(a, b):
    """Renvoie la moyenne de deux nombres a et b."""
    return ............

def est_majeur(age):
    """Renvoie True si la personne est majeure, False sinon."""
    return ............
```

③ `print` ou `return` ?

Remarque 2 (Une distinction fondamentale). `print` et `return` se ressemblent mais ne font **pas la même chose** :

— `print(x)` *affiche* la valeur de `x` à l'écran. La valeur n'est plus accessible ensuite.
— `return x` *renvoie* la valeur de `x` comme résultat de la fonction. La valeur peut être affectée à une variable, passée à une autre fonction, utilisée dans un calcul, etc.

Une fonction qui ne contient que des `print` est rarement réutilisable ; une fonction qui `return` se compose facilement avec d'autres.

Exercice 4 *Repérer la différence*

Voici deux versions d'une fonction :

```python
# Version A
def carre_a(x):
    print(x * x)

# Version B
def carre_b(x):
    return x * x
```

1. Pour chaque appel ci-dessous, dire ce qui s'affiche, puis ce que vaut la variable `r` après exécution. Si une erreur se produit, l'expliquer.

a) `r = carre_a(4)`

b) `r = carre_b(4)`

c) `r = carre_a(4) + 1`

d) `r = carre_b(4) + 1`

2. Pourquoi dit-on que la version A est « une impasse » ?

4 Paramètres, arguments et variables locales

Définition 2 **(Variables locales)** Une variable créée à l'intérieur d'une fonction (paramètre ou affectation) est dite **locale** : elle n'existe qu'à l'intérieur de la fonction et disparaît dès que celle-ci se termine.

Exercice 5 *Variables locales en action*

Considérer le code suivant :

```
def calcul(x):
    a = x + 1
    b = a * 2
    return b

resultat = calcul(5)
print(resultat)
print(a)
```

1. Que vaut `resultat` ? Détailler les valeurs successives prises par `a` et `b` pendant l'exécution de `calcul(5)`.
2. La dernière ligne (`print(a)`) déclenche une erreur. Pourquoi ?
3. Modifier le programme pour que la valeur de `a` (à la fin de l'appel) soit visible depuis l'extérieur de la fonction. *Indication : utiliser un `return` renvoyant un `tuple`.*

5 Documenter et sécuriser une fonction

Définition 3 **(Docstring)** Une **docstring** (« chaîne de documentation ») est une chaîne placée juste après l'en-tête `def`, entre triples guillemets. Elle décrit ce que fait la fonction. La fonction `help(nom_fonction)` affiche cette docstring.

Définition 4 **(Précondition et `assert`)** Une **précondition** est une propriété que les arguments doivent vérifier pour que la fonction donne un résultat correct. On peut la *vérifier* avec l'instruction `assert` : si la condition est fausse, le programme s'arrête immédiatement avec une erreur.

```python
def racine_carree(x):
    assert x >= 0, "x doit être positif ou nul"
    return x ** 0.5
```

Exercice 6 *Documenter et protéger une fonction*

On donne la fonction suivante, qui calcule l'année de naissance d'une personne dont on connaît l'âge en 2026 :

```python
def annee_de_naissance(age):
    return 2026 - age
```

1. Ajouter une docstring d'une ligne décrivant ce que fait la fonction.
2. Quelle précondition raisonnable peut-on imposer sur `age` ? L'écrire avec une instruction `assert`.
3. Tester la fonction avec `annee_de_naissance(20)`, puis avec `annee_de_naissance(-5)`. Que se passe-t-il dans chaque cas ?
4. Pourquoi est-il préférable d'utiliser `assert` plutôt que de simplement « faire confiance » à l'utilisateur de la fonction ?

6 Composer plusieurs fonctions

Exercice 7 *Construire un programme à partir de petites fonctions*

On souhaite écrire un programme qui demande à l'utilisateur la longueur et la largeur d'un rectangle, puis affiche son aire et son périmètre.

1. Définir trois fonctions :
   - `aire(L, l)` qui renvoie l'aire d'un rectangle ;
   - `perimetre(L, l)` qui renvoie son périmètre ;
   - `decrire(L, l)` qui **affiche** (avec `print`) un message du type « Un rectangle de 3 sur 4 a pour aire 12 et pour périmètre 14. ».
2. Pourquoi est-il préférable que `aire` et `perimetre` utilisent `return` (et pas `print`) ? Pourquoi `decrire` peut, lui, utiliser `print` ?
3. Écrire le programme principal qui appelle `decrire(3, 4)`.

7 Ce qu'il faut retenir

Propriété 1 **(Définir et appeler)** Une fonction est définie par `def nom(parametres):` suivi d'un bloc indenté terminé (en général) par un `return`. Elle est appelée par `nom(arguments)`, ce qui produit une valeur (la valeur renvoyée).

Propriété 2 **(`print` vs `return`)** `print` *affiche* sans rien renvoyer ; `return` *renvoie* sans rien afficher. Privilégier `return` pour qu'une fonction reste composable.

Propriété 3 **(Variables locales)** Toute variable créée à l'intérieur d'une fonction est locale : elle disparaît à la fin de l'appel. Pour communiquer une valeur à l'extérieur, il faut la `return`.

Propriété 4 **(Préconditions et documentation)** Une fonction bien écrite :

— commence par une **docstring** décrivant son rôle ;

— vérifie ses **préconditions** avec `assert` pour échouer tôt en cas de mauvais usage ;

— utilise `return` pour produire un résultat réutilisable.