Fonctions en Python

Modularité et réutilisation

Résumé

Les fonctions permettent de découper un programme en sous-programmes réutilisables. Elles évitent la répétition de code, améliorent la lisibilité et facilitent les tests. Ce cours présente la définition, l'appel, les paramètres, la valeur de retour et la portée des variables.

1 Pourquoi des fonctions ?

Supposons que l'on veuille afficher l'aire de plusieurs rectangles :

```
# Sans fonction : code répétitif
aire1 = 5 * 3
print(f"Aire : {aire1}")
aire2 = 8 * 4
print(f"Aire : {aire2}")
aire3 = 12 * 7
print(f"Aire : {aire3}")
```

Si le calcul change, il faut modifier trois lignes. Avec une fonction, on ne modifie qu'à un seul endroit.

2 Définir et appeler une fonction

Définition 1 **(Fonction)** Une **fonction** est un bloc de code nommé, réutilisable, qui :

- peut recevoir des données en entrée (les **paramètres**) ;
- effectue un traitement ;
- peut renvoyer un résultat (via `return`).

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

# Appel de la fonction
resultat = aire_rectangle(5, 3)
print(resultat)  # 15
```

Remarque 1. Le mot clé `def` introduit la **définition**. Le code de la fonction n'est **pas exécuté** à ce moment : il faut un **appel** pour lancer l'exécution.

Exercice 1 *Premières fonctions*

1. Écrivez une fonction `perimetre_rectangle(longueur, largeur)` qui renvoie le périmètre d'un rectangle.

2. Écrivez une fonction `est_pair(n)` qui renvoie `True` si `n` est pair, `False` sinon.

3. Testez : `perimetre_rectangle(5, 3)` doit renvoyer `16` et `est_pair(7)` doit renvoyer `False`.

③ Paramètres et arguments

Les **paramètres** sont les variables déclarées dans la définition. Les **arguments** sont les valeurs fournies lors de l'appel.

```
def saluer(prenom, message):      # paramètres
    print(f"{message}, {prenom} !")

saluer("Alice", "Bonjour")        # arguments
```

Valeurs par défaut

Un paramètre peut avoir une **valeur par défaut** :

```
def saluer(prenom, message="Bonjour"):
    print(f"{message}, {prenom} !")

saluer("Alice")               # Bonjour, Alice !
saluer("Bob", "Bonsoir")    # Bonsoir, Bob !
```

Arguments nommés

On peut préciser le nom des paramètres lors de l'appel :

```
def fiche(nom, age, ville):
    print(f"{nom}, {age} ans, {ville}")

fiche(ville="Lyon", nom="Alice", age=17)
```

Exercice 2 *Paramètres et valeurs par défaut*

1. Écrivez une fonction `puissance(base, exposant=2)` qui renvoie `base` élevé à la puissance `exposant`. Par défaut, elle calcule le carré.

2. Donnez le résultat de :

   a) `puissance(5)`

   b) `puissance(2, 10)`

   c) `puissance(exposant=3, base=4)`

④ `return` vs `print`

C'est une source de confusion fréquente :

```python
def double_print(n):
    print(n * 2)       # affiche, mais ne renvoie rien

def double_return(n):
    return n * 2        # renvoie la valeur

a = double_print(5)    # affiche 10, mais a vaut None
b = double_return(5)   # n'affiche rien, mais b vaut 10
```

Propriétés

— `print()` **affiche** une valeur à l'écran mais la fonction renvoie `None` ;

— `return` **renvoie** une valeur au programme appelant et **termine** la fonction ;

— une fonction sans `return` renvoie automatiquement `None`.

Exercice 3 *Distinguer return et print* Pour chaque code, indiquez ce qui est **affiché** et la valeur de `x` :

1.

```python
def f(n):
    return n + 1

x = f(5)
print(x)
```

2.

```python
def g(n):
    print(n + 1)

x = g(5)
print(x)
```

3.

```python
def h(n):
    n + 1

x = h(5)
print(x)
```

⑤ Portée des variables

Définition 2 **(Portée)** La **portée** d'une variable est la zone du programme où elle est accessible.

— Une variable définie **dans** une fonction est **locale** : elle n'existe que pendant l'exécution de la fonction ;

— une variable définie **en dehors** de toute fonction est **globale** : elle est accessible partout en lecture.

```python
x = 10              # variable globale

def ma_fonction():
    y = 5           # variable locale
    print(x)        # 10 (lecture globale : OK)
    print(y)        # 5

ma_fonction()
print(x)            # 10
print(y)            # ERREUR ! y n'existe pas ici
```

Masquage

Si une variable locale porte le même nom qu'une globale, la locale **masque** la globale :

```python
x = 10

def ma_fonction():
    x = 99          # crée une variable LOCALE x
    print(x)        # 99
```

```
6
7  ma_fonction()
8  print(x)              # 10 (la globale n'a pas changé)
```

Exercice 4 *Portée des variables* Pour chaque programme, indiquez ce qui est affiché :

1.
```
1  a = 5
2  def f():
3      a = 10
4      return a
5  print(f())
6  print(a)
```

2.
```
1  compteur = 0
2  def incrementer():
3      compteur = compteur + 1
4  incrementer()
```

3. Dans le programme 2, pourquoi obtient-on une erreur ? Comment la corriger proprement ?

⑥ Documenter une fonction : la docstring

Une **docstring** est une chaîne placée juste après `def`. Elle documente le rôle, les paramètres et la valeur de retour :

```
1  def moyenne(a, b):
2      """Calcule la moyenne de deux nombres.
3
4      Paramètres :
5          a, b (float) : deux nombres
6      Retour :
7          float : la moyenne de a et b
8      """
9      return (a + b) / 2
```

On la consulte avec `help(moyenne)` ou `moyenne.__doc__`.

Types et exemples dans la documentation

On peut indiquer dans la ligne `def` le type attendu de chaque paramètre et le type de la valeur renvoyée. Ces **annotations** servent de documentation : Python ne les vérifie pas.

```
def moyenne(a: float, b: float) -> float:
    """Calcule et renvoie la moyenne des deux nombres a et b.

    >>> moyenne(12, 15)
    13.5
    """
    return (a + b) / 2
```

`a: float` indique que `a` est un nombre flottant, et `-> float` que la fonction renvoie un flottant. Les types usuels sont `int`, `float`, `str` et `bool`. Les lignes qui commencent par `>>>` dans la docstring montrent un exemple d'appel, tel qu'on le taperait dans la console, suivi du résultat attendu.

⑦ Tester une fonction : `assert`

Une fois la fonction écrite, on vérifie qu'elle renvoie les résultats attendus sur quelques exemples dont on connaît la réponse. L'instruction `assert` teste une condition : si elle est vraie, il ne se passe rien ; si elle est fausse, Python arrête le programme avec une erreur `AssertionError`.

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

assert carre(3) == 9
assert carre(-2) == 4
assert carre(0) == 0
print("Tous les tests sont réussis")
```

Si la fonction contient une erreur, par exemple `return n * 2` au lieu de `return n ** 2`, le premier test échoue : Python affiche `AssertionError` et le programme s'arrête. On place donc les tests juste après la fonction, pour repérer une erreur dès la première exécution.

Exercice 5 *Fonctions complètes* Écrivez les fonctions suivantes avec une docstring et des assertions de test :

1. `factorielle(n)` qui renvoie $n! = 1 \times 2 \times \cdots \times n$ (avec $0! = 1$).

2. `est_premier(n)` qui renvoie `True` si $n \geq 2$ est premier, `False` sinon.

Synthèse

Concept	Description
`def f(params):`	Définit une fonction nommée `f`
`return valeur`	Renvoie une valeur et termine la fonction
`print()`	Affiche à l'écran mais ne renvoie rien (`None`)
Valeur par défaut	`def f(x, y=0):` — `y` vaut 0 si non précisé
Variable locale	Définie dans la fonction, invisible à l'extérieur
Variable globale	Définie hors fonction, lisible partout
Masquage	Une locale de même nom cache la globale
Docstring	Chaîne documentant la fonction, après `def`
Annotations	`def f(x: int) -> int:` indique les types, sans les vérifier
`assert condition`	Arrête le programme si la condition est fausse : sert à tester une fonction