

Programmation en Python - Séance 5 : Création de Fonctions

Hervé TALE KALACHI

ENSPY – Université de Yaoundé 1

Définition. Une **fonction** est un **bloc d'instructions réutilisable** qui exécute une tâche précise et peut renvoyer un résultat.

Pourquoi utiliser des fonctions ?

- Éviter la duplication de code (réutilisation).
- Faciliter la lecture et la maintenance.
- Découper un grand problème en sous-problèmes.
- Tester et corriger plus facilement.

Exemples typiques :

- Calculer une moyenne.
- Trouver le maximum d'une liste.
- Gérer une interaction utilisateur.

Déclaration :

```
def nom_fonction(param1, param2, ...):  
    """Documentation (optionnelle)."""  
    bloc_d_instructions  
    return resultat # optionnel
```

Exemple simple :

```
def saluer(nom):  
    print("Bonjour", nom)  
  
saluer("Alice") # Appel de la fonction
```

Règles de base :

- Le mot-clé `def` introduit une fonction.
- Les paramètres sont optionnels.
- Le mot-clé `return` renvoie une valeur au programme principal.

Exemple :

```
def somme(a, b):  
    return a + b  
  
resultat = somme(3, 5)  
print("Résultat =", resultat)
```

Types d'arguments :

- **Positionnels** : selon l'ordre des paramètres.
- **Només** : précisés par leur nom (`somme(a=3, b=5)`).
- **Valeurs par défaut** :

```
def dire_bonjour(nom="inconnu"):  
    print("Bonjour", nom)
```

Astuce : combinez arguments nommés et valeurs par défaut pour plus de souplesse.

Objectif : produire un résultat réutilisable ailleurs dans le programme.

```
def aire_rectangle(longueur, largeur):  
    aire = longueur * largeur  
    return aire  
  
resultat = aire_rectangle(4, 2)  
print("Aire =", resultat)
```

Remarques :

- `return` interrompt la fonction et renvoie une valeur.
- Une fonction sans `return` renvoie implicitement `None`.

À tester : que se passe-t-il si vous retirez `return` dans le code ?

Les fonctions — portée des variables

Portée : une variable n'existe que dans la zone où elle est définie.

```
x = 10 # variable globale

def test():
    x = 5 # variable locale
    print("x local =", x)

test()
print("x global =", x)
```

Résultat :

```
x local = 5
x global = 10
```

Règles :

- Les variables définies à l'intérieur d'une fonction sont locales.
- Pour modifier une variable globale, utiliser le mot-clé `global`.

1) Calcul de la moyenne :

```
def moyenne(notes):  
    return sum(notes) / len(notes)  
  
print(moyenne([12, 14, 16]))
```

2) Vérifier si un nombre est pair :

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

3) Affichage formaté :

```
def afficher_etudiant(nom, age, moyenne):  
    print(f"{nom} ({age} ans) - Moyenne : {moyenne:.2f}")
```

À vous de jouer — exercices sur les fonctions

- # A) *Creez une fonction `carre(x)` qui renvoie le carre d'un nombre .*
- # B) *Creez une fonction `maximum(a, b)` qui renvoie la plus grande valeur.*
- # C) *Creez une fonction `factorielle(n)` qui calcule $n!$ avec une boucle.*
- # D) *Creez une fonction `convert(celsius)` qui renvoie la temperature en F.*
- # E) *Creez une fonction `compte_voyelles(chaine)`
qui renvoie le nombre de voyelles dans la chaîne.*

Astuce : testez-les une par une dans un fichier `fonctions.py` ou en mode interactif.

Définition. `*args` et `**kwargs` permettent de passer un nombre variable d'arguments à une fonction.

1) `*args` — liste d'arguments positionnels :

```
def somme(*args):  
    return sum(args)  
  
print(somme(2, 4, 6)) # 12  
print(somme(1, 2, 3, 4, 5)) # 15
```

2) `**kwargs` — dictionnaire d'arguments nommés :

```
def afficher_infos(**kwargs):  
    for cle, val in kwargs.items():  
        print(f"{cle} : {val}")  
  
afficher_infos(nom="Alice", age=22, ville="Yaoundé")
```

Ordre des paramètres dans une fonction

Règle d'ordre :

- 1 Paramètres positionnels obligatoires
- 2 Paramètres avec valeurs par défaut
- 3 `*args`
- 4 `**kwargs`

```
def exemple(a, b=0, *args, **kwargs):  
    print("a =", a)  
    print("b =", b)  
    print("args =", args)  
    print("kwargs =", kwargs)  
  
exemple(10, 20, 30, 40, x=5, y=7)
```

Fonctions anonymes — lambda

Définition. Une lambda est une fonction courte et anonyme, utile pour des calculs simples ou temporaires.

Syntaxe :

```
lambda arguments: expression
```

Exemples :

```
carre = lambda x: x**2  
print(carre(5)) # 25
```

```
somme = lambda a, b: a + b  
print(somme(2, 3)) # 5
```

Cas d'usage :

```
nombres = [3, 1, 4, 2]  
nombres.sort(key=lambda x: -x)  
print(nombres) # [4, 3, 2, 1]
```

Définition. Une docstring est une chaîne de caractères placée juste après la déclaration de la fonction, servant de documentation intégrée.

Exemple :

```
def carre(x):  
    """Renvoie le carré de x."""  
    return x**2  
  
help(carre)  
print(carre.__doc__)
```

Bonnes pratiques :

- Résumer le rôle de la fonction.
- Expliquer les paramètres et la valeur de retour.
- Employer un style concis et informatif.

PEP 8 = Guide de style officiel de Python. Voici les principales recommandations pour les fonctions :

- Utiliser des noms en minuscules avec des underscores : `def aire_cercle(rayon):`
- Limiter la longueur des lignes à 79 caractères.
- Mettre deux lignes vides avant chaque définition de fonction.
- Indenter avec 4 espaces, jamais de tabulations.
- Ajouter une docstring si la fonction n'est pas triviale.
- Respecter l'ordre logique du code : définitions → exécution → tests éventuels.

But : produire un code propre, lisible et cohérent entre tous les développeurs.

Objectif : combiner paramètres, valeur par défaut et `*args`.

```
def moyenne_ponderee(*notes, coeff=1):  
    """Calcule la moyenne pondérée des notes."""  
    if not notes:  
        return 0  
    return sum(notes) / len(notes) * coeff  
  
print(moyenne_ponderee(10, 12, 14))  
print(moyenne_ponderee(10, 12, 14, coeff=1.5))
```

À vous de jouer :

- Ajoutez un paramètre `arrondi=False` pour arrondir le résultat.
- Testez avec plusieurs listes de notes.

À vous de jouer — exercices sur les fonctions avancées

```
# A) Créez une fonction afficher_infos(**kwargs)
# qui affiche chaque clé et valeur sur une ligne.

# B) Créez une fonction somme_pairs(*args)
# qui additionne uniquement les nombres pairs.

# C) Créez une fonction comparer(x, y, *, inverse=False)
# qui renvoie le plus grand (ou le plus petit si inverse=True).

# D) Créez une fonction carre = lambda x: x**2
# puis testez-la dans une boucle for.

# E) Ajoutez une docstring claire à chacune de vos fonctions.
```

Astuce : utilisez le REPL pour tester les fonctions avant de les regrouper dans un script.