TP Chapitre 2 : Les Fonctions

Objectifs du TP

Objectifs du TP. À l'issue de cette séance, tu dois être capable de :

- expliquer ce qu'est une fonction : un bloc d'instructions nommé, avec des paramètres et une valeur renvoyée ;
- définir tes propres fonctions avec `def` et `return` ;
- distinguer *renvoyer* une valeur (`return`) et *afficher* une valeur (`print`) ;
- écrire une fonction qui appelle une autre fonction, pour réutiliser du code déjà écrit ;
- utiliser des fonctions prédéfinies (`int`, `float`, `str`, `len`, `abs`, `round`) et des conversions de types ;
- importer et utiliser des bibliothèques (`math`, `random`) ;
- concevoir un programme complet organisé autour de fonctions à partir d'une spécification (entrées, traitement, sorties).

Consignes générales. Tous les exercices se traitent en Python (console ou fichier). Enregistre régulièrement ton travail. Quand un programme ne fonctionne pas, lis le message d'erreur : il indique souvent la ligne et le type d'erreur. La partie B se fait *en autonomie* : tu écris des programmes entiers à partir d'une spécification, aucune ligne de code n'est fournie.

Partie A — Application guidée

Les exercices suivants se programment en Python. Ils sont classés du plus simple au plus difficile. Chaque exercice se termine par un *point de validation* (encadré vert) : le résultat exact que doit produire ton programme pour être juste. Complète les squelettes de code en respectant l'indentation.

Exercice A1 — Prix TTC (ma première fonction)

Un magasin applique la TVA à 20 % sur le prix hors taxes : le prix TTC vaut prix HT
times 1,2. On veut définir une fonction `prix_ttc(prix_ht)` qui *renvoie* le prix TTC, puis l'utiliser pour deux achats. Compléter le squelette suivant.

```
def prix_ttc(prix_ht):
    # A COMPLETER : renvoyer le prix TTC (TVA a 20 %)

print(prix_ttc(80))
print(prix_ttc(120))
```

Point de validation

Le programme doit afficher `96.0` puis `144.0`.

Exercice A2 — La remise en soldes (fonction à deux paramètres)

Pendant les soldes, un magasin applique une remise en pourcentage sur tous les articles : le prix soldé vaut prix
times (1 − taux/100). Écrire une fonction `remise_solde(prix, taux)` qui renvoie le prix après remise. Compléter le squelette.

```
def remise_solde(prix, taux):
    # A COMPLETER : renvoyer le prix apres remise
```

```
3
4 print(remise_solde(80, 20))
5 print(remise_solde(50, 10))
```

Point de validation

Le programme doit afficher `64.0` puis `45.0`.

Exercice A3 — La TVA réutilisée (une fonction qui en appelle une autre)

On veut maintenant séparer le calcul de la TVA de celui du prix TTC. La fonction `tva(prix_ht)` est déjà écrite : elle renvoie le montant de la TVA (20 % du prix HT). Compléter la fonction `prix_ttc(prix_ht)` : elle doit renvoyer `prix_ht + tva(prix_ht)`, c'est-à-dire *appeler* la fonction `tva`.

```
1 def tva(prix_ht):
2     return prix_ht * 0.2
3
4 def prix_ttc(prix_ht):
5     # A COMPLETER : renvoyer prix_ht + tva(prix_ht)
6
7 print(prix_ttc(80))
8 print(prix_ttc(150))
```

Point de validation

Le programme doit afficher `96.0` puis `180.0`.

Exercice A4 — Le tarif dégressif (plusieurs `return` avec une condition)

Un grossiste vend des cartouches d'encre : le prix unitaire dépend de la quantité commandée :

— moins de 10 cartouches : 5 € pièce ;

— de 10 à 49 cartouches : 4 € pièce ;

— 50 cartouches ou plus : 3 € pièce.

Écrire une fonction `tarif_unitaire(quantite)` qui renvoie le prix unitaire correspondant, à l'aide d'une instruction conditionnelle (un `return` dans chaque cas). Compléter le squelette.

```
1 def tarif_unitaire(quantite):
2     # A COMPLETER : trois cas avec if / elif / else
3
4 print(tarif_unitaire(5))
5 print(tarif_unitaire(10))
6 print(tarif_unitaire(50))
```

Point de validation

Le programme doit afficher `5` puis `4` puis `3`.

Partie B — En autonomie

Tu écris maintenant des programmes *entiers*, sans squelette. Pour chaque exercice : lis la spécification (entrées, traitement, sorties, contraintes), réponds d'abord aux questions de conception de l'encadré orange (sur papier ou à voix haute), puis écris ton programme dans un fichier et teste-le avec les

valeurs de l'encadré vert. L'objectif est de concevoir toi-même la solution : ne demande pas de lignes de code toutes faites.

Exercice B1 — Le budget mensuel (fonction et programme principal)

Chaque mois, un étudiant reçoit son budget (argent de poche, petit boulot…) et doit payer ses dépenses fixes : loyer, transport, nourriture. Il veut savoir ce qui lui reste à la fin du mois.

Énoncé. Écrire un programme qui demande le budget, le loyer, le transport et la nourriture, calcule le reste à l'aide d'une fonction `reste_budget(budget, loyer, transport, nourriture)`, puis affiche le reste et indique si le budget est respecté ou dépassé.

Spécification.

— *Entrées* : budget mensuel, loyer, transport, nourriture (quatre flottants, en euros).

— *Traitement* : la fonction renvoie budget − (loyer + transport + nourriture) ; le programme principal appelle la fonction et teste le signe du reste.

— *Sorties* : `Reste : ... euros`, puis `Budget respecte` si le reste est positif ou nul, `Budget depasse` sinon.

— *Contraintes* : la fonction doit *renvoyer* le reste, elle ne doit pas l'afficher ; le test du signe se fait dans le programme principal.

Pour t'aider — questions de conception (sans donner le code)

— Que doit renvoyer la fonction ? Pourquoi ne doit-elle pas afficher elle-même le résultat ?

— Quels sont les paramètres de la fonction ? Dans quel ordre les écrire ?

— Pourquoi faut-il écrire `float(input(...))` et pas `input(...)` ?

— Que doit afficher le programme si le reste vaut exactement 0 ?

Point de validation

Vérifie : budget 500, loyer 300, transport 40, nourriture 100
rightarrow `Reste : 60.0 euros` puis `Budget respecte` ; budget 400 avec les mêmes dépenses

rightarrow `Reste : -40.0 euros` puis `Budget depasse`.

Exercice B2 — L'épargne programmée (fonction avec boucle `for`)

Tu épargnes une somme fixe chaque mois sur un livret sans intérêts, pour atteindre un objectif (vacances, ordinateur, permis…).

Énoncé. Écrire une fonction `total_epargne(apport, nb_mois)` qui renvoie le total versé après `nb_mois` mois, à l'aide d'une boucle `for` et d'un accumulateur. Le programme demande l'apport, le nombre de mois et un objectif ; il appelle la fonction et affiche le total versé, puis indique si l'objectif est atteint ou non.

Spécification.

— *Entrées* : apport mensuel (flottant positif), nombre de mois (entier *geq*0), objectif (flottant positif).

— *Traitement* : accumulateur dans la fonction (boucle `for`) ; comparaison total *geq* objectif dans le programme principal.

— *Sorties* : `Total verse : ... euros`, puis `Objectif atteint` ou `Objectif non atteint`.

— *Contraintes* : la boucle `for` est *à l'intérieur* de la fonction ; la fonction renvoie le total ; le programme principal ne contient ni boucle ni accumulateur pour le total.

Pour t'aider — questions de conception (sans donner le code)

— Où initialiser l'accumulateur ? Pourquoi ?

— Pourquoi une boucle `for` est-elle adaptée ici ? (combien de tours sait-on à l'avance ?)

— Quel est l'intérêt de mettre la boucle dans une fonction plutôt que dans le programme principal ?

— Comment tester la fonction séparément du reste du programme ?

Point de validation

Vérifie : apport 100, 12 mois, objectif 500
rightarrow `Total verse : 1200.0 euros` puis `Objectif atteint` ; apport 100, 12 mois, objectif 2000
rightarrow `Total verse : 1200.0 euros` puis `Objectif non atteint`.

Exercice B3 — Le distributeur vérifié (fonction booléenne et boucle de saisie)

On reprend le distributeur de billets du chapitre 1 : un montant est valide s'il est strictement positif et multiple de 10.

Énoncé. Écrire une fonction `montant_valide(montant)` qui renvoie `True` si le montant est valide et `False` sinon. Le programme demande un montant ; tant que la fonction renvoie `False`, il affiche un message d'erreur et redemande ; quand le montant est valide, il le décompose en billets de 50 €, 20 € et 10 € avec le moins de billets possible et affiche le résultat.

Spécification.

— *Entrées* : un montant (entier) saisi au clavier, éventuellement plusieurs fois.

— *Traitement* : la fonction teste la validité (montant > 0 et montant $\%$ 10 $==$ 0) ; boucle `while` dont la condition utilise l'appel de la fonction ; décomposition avec `//` et `%`.

— *Sorties* : un message d'erreur à chaque saisie invalide ; puis `Billets de 50 : ...`, `Billets de 20 : ...`, `Billets de 10 : ...`.

— *Contraintes* : la fonction renvoie un booléen ; la condition de la boucle `while` est un appel à la fonction ; l'opérateur `and` est autorisé.

Pour t'aider — questions de conception (sans donner le code)

— Quel type de valeur renvoie la fonction ? Comment s'en servir dans la condition du `while` ?

— Comment traduire « strictement positif et multiple de 10 » en Python ?

— Pourquoi mettre le test dans une fonction : quel avantage par rapport à un simple `if` dans le programme principal ?

— Que se passerait-il si on oubliait le cas « strictement positif » ? (teste mentalement avec 0)

Point de validation

Vérifie : `montant_valide(120)`
rightarrow `True` ; `montant_valide(25)`
rightarrow `False` ; `montant_valide(0)`
rightarrow `False`. Le programme complet avec 120
rightarrow 2 billets de 50, 1 de 20, 0 de 10 ; avec 25

rightarrow message d'erreur puis nouvelle saisie.

Exercice B4 — Le jeu de pile ou face (bibliothèque `random`)

Un jeu de hasard : on lance une pièce. Si elle tombe sur pile, le joueur gagne 2 €; sur face, il perd 1 €. On veut simuler 100 parties et afficher le gain total (qui peut être négatif).

Énoncé. Écrire un programme qui :

1. définit une fonction `lancer_piece()` qui renvoie `"pile"` ou `"face"` en utilisant `rd.randint` ;
2. définit une fonction `gain(face)` qui renvoie 2 si `face` vaut `"pile"` et -1 sinon ;
3. joue 100 parties : à chaque partie, appel de `lancer_piece()`, calcul du gain, accumulation ; à la fin, affiche le gain total.

Spécification.

— *Entrées* : aucune (le nombre de parties est fixé à 100).

— *Traitement* : `import random as rd` en début de programme ; 100 parties ; à chaque partie, appel de `lancer_piece()` puis de `gain(...)` ; accumulateur.

— *Sorties* : le gain total après les 100 parties (entier, éventuellement négatif).

— *Contraintes* : les deux fonctions renvoient des valeurs ; le programme principal utilise les deux fonctions ; la fonction `gain` ne doit pas afficher elle-même.

Pour t'aider — questions de conception (sans donner le code)

— Que renvoie `rd.randint(0, 1)` ? Comment relier 0 et 1 à « pile » et « face » ?

— Pourquoi la fonction `gain` doit-elle *renvoyer* un nombre et pas seulement l'afficher ?

— Quel est l'intérêt de séparer `lancer_piece()` et `gain(face)` en deux fonctions ?

— Le gain total peut-il être négatif ? Que doit afficher le programme dans ce cas ?

Point de validation

Vérifie : `gain("pile")`
rightarrow `2` ; `gain("face")`
rightarrow `-1` ; `lancer_piece()` renvoie toujours `"pile"` ou `"face"`. Le résultat du programme est aléatoire : sur 100 parties, le gain total est un entier, en général compris entre 10 et 90 (l'espérance est de 0,5 € par partie, soit 50 € en moyenne).

Partie C — Exercice de réflexion (à finir à la maison)

À finir à la maison

Mon portefeuille d'actions : simulation d'un investissement en bourse.

Contexte. Tu investis 1000 € dans des actions. Chaque jour, la valeur de ton portefeuille varie d'un pourcentage entier aléatoire compris entre -2 % et $+2$ % (les valeurs -2, -1, 0, 1, 2 sont équiprobables) : on multiplie la valeur du jour par $1 +$
$fracp100$, où p est le pourcentage tiré au hasard.

Objectif. Écrire un programme qui simule l'évolution du portefeuille sur N jours (N saisi par l'utilisateur), puis affiche : la valeur finale du portefeuille, le gain ou la perte en euros (valeur finale $-$ 1000), et le rendement en pourcentage (gain
div 1000

times 100). Le programme doit être organisé autour de fonctions.

Questions à se poser avant de coder.

— Quelles fonctions sont utiles ? Que doit renvoyer chacune ? (proposer au moins : une fonction qui tire le pourcentage du jour, une fonction qui calcule la valeur après un jour)

— Quelle bibliothèque importer pour le tirage aléatoire ? Comment tirer un entier entre -2 et 2 ?

— Quelle boucle utiliser pour simuler les N jours ? Que doit-on mettre à jour à chaque tour ?

— Comment vérifier que le programme est juste ? (penser à un cas simple : remplacer le tirage aléatoire par un pourcentage fixe de 0 % : la valeur doit rester 1000)

— Le rendement peut-il être négatif ? Que doit afficher le programme dans ce cas ?

Étapes suggérées.

1. Version 0 : fonction `variation()` qui renvoie un entier aléatoire entre -2 et 2 (bibliothèque `random`) ; la tester en l'affichant plusieurs fois.
2. Version 1 : fonction `valeur_apres_jour(valeur)` qui applique une variation au portefeuille ; tester avec un pourcentage fixe de 0 % : la valeur ne change pas.
3. Version 2 : boucle sur N jours avec mise à jour de la valeur ; afficher la valeur finale.
4. Version 3 : calculer le gain et le rendement, et afficher un bilan complet.
5. Bonus : lancer plusieurs simulations de 100 jours et comparer les valeurs finales (pourquoi diffèrent-elles ?) ; ajouter une règle de « stop-loss » : si la valeur descend sous 900 €, on vend tout et la simulation s'arrête.

Contraintes. Investissement initial de 1000 € ; nombre de jours entre 1 et 365 ; programme organisé autour d'au moins deux fonctions ; programme testé et commenté ; les messages peuvent être écrits sans accents (ex. `Valeur`, `Gain`) si ton éditeur pose problème.

Fin du TP — la partie C est à finir à la maison.