TP Chapitre 6 : Modularité

Objectifs du TP

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

- expliquer ce qu'est une **API** et comment on l'interroge par une URL ;
- lire et exploiter des données au format **JSON** (dictionnaires imbriqués) ;
- récupérer des données avec le module `requests` et les convertir avec `r.json()` ;
- créer un **module** Python et le documenter avec des **docstrings** ;
- importer un module proprement (`import module`, `from module import f`, `import module as m`) et éviter `from module import *` ;
- distinguer l'**interface** d'un module de son **implémentation** (boîte noire) ;
- concevoir et écrire un programme complet à partir d'une spécification (entrées, traitement, sorties).

Consignes générales. Dans ce TP, tu crées plusieurs fichiers Python : des *modules* (fichiers `.py`) et des *programmes* qui les utilisent. Place tous les fichiers d'un même exercice dans le même dossier : un module s'importe par son **nom**, c'est-à-dire le nom du fichier sans l'extension `.py`. 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 — Le taux de change en JSON (dictionnaires imbriqués)

Une API de taux de change renvoie ses données au format **JSON**, qui se lit comme un dictionnaire Python. Voici un extrait de la réponse (données d'exemple) :

```
{"base": "EUR", "rates": {"USD": 1.09, "GBP": 0.85}}
```

La clé `"rates"` est associée à un dictionnaire contenant les taux de change par rapport à l'euro : 1 euro vaut 1,09 dollar et 0,85 livre. Complète le squelette pour afficher la devise de base, le taux du dollar, le taux de la livre, puis le montant en dollars correspondant à 30 €.

```
taux = {"base": "EUR", "rates": {"USD": 1.09, "GBP": 0.85}}
montant = 30

print("Devise de base :", taux["base"])
print("Taux USD :", taux["rates"]["USD"])
# A COMPLETER : afficher le taux GBP (meme principe)
# A COMPLETER : afficher le montant converti en dollars (montant * taux USD)
```

Point de validation

Vérifie ton programme : il doit afficher, dans l'ordre, `Devise de base : EUR`, `Taux USD : 1.09`, `Taux GBP : 0.85` et `30 euros en dollars : 32.7`.

Exercice A2 — Module `tva` : prix TTC et TVA (créer un module documenté)

Un commerce veut réutiliser ses calculs de TVA (20 %) dans plusieurs programmes. On regroupe ces calculs dans un **module** `tva.py` : un fichier qui ne contient que des définitions de fonctions, documentées par des **docstrings** (chaîne entre triple guillemets qui décrit le rôle de la fonction).

Complète le fichier `tva.py` : chaque fonction doit renvoyer le bon résultat.

```
def ttc(prix_ht):
    """Renvoie le prix TTC : prix HT augmente de la TVA a 20%."""
    # A COMPLETER : calcul du prix TTC
    return 0.0

def tva(prix_ht):
    """Renvoie le montant de la TVA : 20% du prix HT."""
    # A COMPLETER : calcul du montant de la TVA
    return 0.0
```

Complète ensuite le programme de test `test_tva.py`, qui importe les deux fonctions du module et affiche trois résultats.

```
from tva import ttc, tva

# A COMPLETER : afficher le prix TTC de 100 euros
# A COMPLETER : afficher la TVA de 100 euros
# A COMPLETER : afficher le prix TTC de 35.50 euros
```

Point de validation

Le programme doit afficher `Prix TTC de 100 euros : 120.0`, `TVA de 100 euros : 20.0` et `Prix TTC de 35.50 euros : 42.6` (libellés libres, valeurs exactes).

Exercice A3 — Module `devises` : conversion de devises (fonctions et docstrings)

Pour préparer un voyage, tu crées le module `devises.py` contenant deux fonctions de conversion. Le taux de change est donné *pour 1 euro* : si le taux vaut 1,09, un montant en euros se convertit en dollars en le *multipliant* par 1,09, et un montant en dollars se convertit en euros en le *divisant* par 1,09. Complète les deux fonctions, puis le programme de test.

```
def vers_euro(montant, taux):
    """Convertit un montant en devise vers des euros (taux = valeur de 1
       euro)."""
    # A COMPLETER : montant divise par le taux
    return 0.0

def depuis_euro(montant, taux):
    """Convertit un montant en euros vers une devise."""
    # A COMPLETER : montant multiplie par le taux
    return 0.0
```

```
from devises import vers_euro, depuis_euro

# A COMPLETER : afficher 100 euros en dollars avec un taux de 1.09
# A COMPLETER : afficher 109 dollars en euros avec un taux de 1.09
```

Pour l'affichage des montants, arrondis à 2 décimales avec `round(montant, 2)` : on manipule de l'argent.

Point de validation

Le programme doit afficher `109.0` puis `100.0` : on repasse par l'euro sans perte.

Exercice A4 — L'API de taux de change avec le module `requests` (lecture JSON)

Une API de taux de change expose ses données à l'adresse suivante (URL de démonstration) :

```
https://samples.api-taux.fr/latest?base=EUR&symbols=USD&appid=0123456789abcdef0123456789abcdef
```

L'URL contient des paramètres : `base` (la devise de référence, ici `EUR`), `symbols` (la devise cible, ici `USD`) et `appid` (la clé de démonstration). Elle renvoie le JSON suivant (données de démonstration figées) :

```
{
  "base": "EUR",
  "date": "2026-01-15",
  "rates": {"USD": 1.09}
}
```

Complète le squelette : après avoir récupéré le dictionnaire avec la fonction `get_taux`, affiche la devise de base puis le taux EUR vers USD.

```
import requests

API_KEY = "0123456789abcdef0123456789abcdef"

def get_taux(api_key, base, cible):
    url = f"https://samples.api-taux.fr/latest?base={base}&symbols={cible}&appid={api_key}"
    r = requests.get(url)
    return r.json()

# A COMPLETER : appeler get_taux pour la paire EUR vers USD
# A COMPLETER : afficher la devise de base
# A COMPLETER : afficher le taux EUR vers USD
```

Pour information : le module `requests` n'appartient pas à la bibliothèque standard de Python ; s'il n'est pas installé, demande à ton professeur ou installe-le avec la commande `pip install requests` (à faire une seule fois). Si le réseau n'est pas disponible, tu peux tester ton programme en remplaçant temporairement l'appel à `get_taux` par le dictionnaire de démonstration ci-dessus.

Point de validation

Avec les données de démonstration, le programme doit afficher `Base : EUR` puis `Taux EUR -> USD : 1.09`.

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 module `budget` : un outil réutilisable

Tu veux gérer ton budget mensuel et le réutiliser chaque mois sans réécrire les calculs.

Énoncé. Écrire un module `budget.py` contenant trois fonctions documentées :

— `total_depenses(liste)` : renvoie la somme des dépenses d'une liste ;

— `solde(entrees, depenses)` : renvoie la différence entre les revenus et les dépenses ;

— `budget_annuel(mensuel)` : renvoie le budget annuel correspondant à un revenu mensuel (revenu $\times$ 12).

Le module commence par une docstring de présentation générale. Écrire ensuite un programme `test_budget.py` qui demande le nombre de dépenses du mois, saisit chaque dépense, demande les revenus du mois, puis affiche le total des dépenses, le solde du mois et le budget annuel estimé.

Spécification.

— *Entrées* : un entier N (nombre de dépenses, $N \geq 0$), puis N montants de dépenses (flottants, en euros), puis les revenus du mois (flottant).

— *Traitement* : total = somme des N dépenses (accumulateur dans `total_depenses`) ; solde = revenus − total ; budget annuel = revenus $\times$ 12.

— *Sorties* : `Total des depenses : ... euros`, `Solde du mois : ... euros`, `Budget annuel estime : ... euros`.

— *Contraintes* : module `budget.py` (docstring générale + docstrings par fonction) ; programme `test_budget.py` avec import ciblé (`from budget import ...`) ; boucle `for` avec accumulateur pour la somme.

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

— Pourquoi placer les trois calculs dans un module plutôt que directement dans le programme de test ?

— Dans quelle fonction trouve-t-on l'accumulateur ? Où doit-il être initialisé et pourquoi ?

— Pourquoi préférer un import ciblé (`from budget import total_depenses, ...`) à `from budget import *` ?

— Comment vérifier le rôle d'une fonction du module sans écrire le programme ? (pense à la fonction `help`)

Point de validation

Vérifie ton programme : avec 3 dépenses à 350, 120 et 45,5 € et 1500 € de revenus, il doit afficher `Total des depenses : 515.5 euros`, `Solde du mois : 984.5 euros` et `Budget annuel estime : 18000.0 euros`.

Exercice B2 — Le convertisseur de devises pour le voyage

Tu pars en voyage aux États-Unis, au Royaume-Uni et en Suisse. Tu veux un programme qui affiche la conversion de ton budget dans les trois devises.

Énoncé. Écrire un module `devises.py` contenant trois fonctions documentées :

— `vers_euro(montant, taux)` : convertit un montant en devise vers des euros ;

— `depuis_euro(montant, taux)` : convertit un montant en euros vers une devise ;

— `afficher_conversion(montant_euros, taux, nom_devise)` : affiche une ligne de conversion.

Écrire ensuite un programme `voyage.py` qui demande le budget en euros, puis affiche sa conversion pour chaque devise du dictionnaire `taux = {"USD": 1.09, "GBP": 0.85, "CHF": 0.94}`.

Spécification.

— *Entrées* : un budget en euros (flottant strictement positif).

— *Traitement* : pour chaque devise du dictionnaire, calculer budget × taux, arrondi à 2 décimales.

— *Sorties* : une ligne par devise, au format `100.0 EUR = 109.0 USD`.

— *Contraintes* : module `devises.py` documenté (docstring générale + docstrings) ; programme `voyage.py` avec import ciblé ; boucle `for` pour parcourir le dictionnaire ; arrondi `round(..., 2)` pour l'affichage.

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

— Que produit la boucle `for devise in taux` ? (que vaut `devise` à chaque tour ?)

— Pourquoi arrondir les montants à 2 décimales avant l'affichage ? (pense aux flottants : que donne `0.1 + 0.2` ?)

— Quelle est l'interface de la fonction `depuis_euro` ? (paramètres, valeur renvoyée)

— Pourquoi le programme `voyage.py` ne doit-il contenir aucune formule de conversion ?

Point de validation

Vérifie ton programme : avec un budget de 100, il doit afficher `100.0 EUR = 109.0 USD`, `100.0 EUR = 85.0 GBP` et `100.0 EUR = 94.0 CHF`.

Exercice B3 — Le panier d'achat récupéré par API (scraping de prix)

Une boutique en ligne de démonstration expose les prix de ses articles via une API. L'URL suivante renvoie le JSON ci-dessous (données de démonstration figées) :

`https://samples.api-prix.fr/panier?appid=0123456789abcdef0123456789abcdef`

```
{
  "boutique": "DemoShop",
  "articles": [
    {"nom": "cahier", "prix_ht": 2.5},
    {"nom": "stylo", "prix_ht": 1.2},
    {"nom": "sac", "prix_ht": 24.0}
  ]
}
```

Énoncé. Écrire un programme `panier.py` qui récupère ces données avec `requests`, affiche le nom et le prix hors taxes de chaque article, calcule le total hors taxes, la TVA (20 % du total HT), le total TTC, puis indique si le total TTC dépasse le budget fixé à 30 €.

Spécification.

— *Entrées* : aucune saisie ; les données sont récupérées depuis l'URL ; budget fixé à 30 dans le programme.

— *Traitement* : pour chaque dictionnaire de la liste `"articles"`, extraire `"nom"` et `"prix_ht"` ; total HT par accumulation ; TVA = total HT × 0,2 ; TTC = total HT × 1,2 (arrondis à 2 décimales) ; comparer TTC à 30.

— *Sorties* : une ligne par article (`nom : prix euros`), puis `Total HT`, `TVA`, `Total TTC` et `Dans le budget` ou `Hors budget`.

— *Contraintes* : `requests` pour récupérer le JSON ; la clé `"articles"` contient une *liste* de dictionnaires ; accumulateur pour le total ; arrondi `round(..., 2)` ; si le réseau est indisponible, remplacer temporairement `r.json()` par le dictionnaire fourni.

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

- Comment écrire l'expression qui donne le prix du premier article ? (la structure est imbriquée)
- Quelle boucle permet de parcourir tous les articles ? Que vaut la variable de boucle à chaque tour ?
- Pourquoi le programme doit-il rester correct si la liste contient 5 articles au lieu de 3 ?
- Que se passe-t-il si le site de démonstration ne répond plus ? (réponds en une phrase, sans coder)

Point de validation

Le programme doit afficher `cahier : 2.5 euros`, `stylo : 1.2 euros`, `sac : 24.0 euros`, `Total HT : 27.7 euros`, `TVA : 5.54 euros`, `Total TTC : 33.24 euros`, puis `Hors budget`.

Exercice B4 — Le tableau de conversion multi-devises (recherche du taux maximal)

Pour choisir une destination, tu compares les taux de change de plusieurs devises. L'API de démonstration de l'exercice A4 renvoie cette fois plusieurs devises :

```
{
  "base": "EUR",
  "date": "2026-01-15",
  "rates": {"USD": 1.09, "GBP": 0.85, "CHF": 0.94, "JPY": 158.4}
}
```

Énoncé. Écrire un programme `tableau.py` qui récupère les taux avec `requests` (URL : `https://samples.api-ta` demande un budget en euros, affiche la conversion du budget pour chaque devise, puis indique la devise dont le taux est le plus élevé.

Spécification.

- *Entrées* : un budget en euros (flottant strictement positif) ; taux récupérés depuis l'URL de démonstration.
- *Traitement* : pour chaque devise du dictionnaire `"rates"`, conversion budget × taux (arrondi à 2 décimales) ; parcours des clés pour trouver la devise de taux maximal (comparaison, sans utiliser `max()`).
- *Sorties* : une ligne de conversion par devise, puis `Taux le plus eleve : X ( taux )`.
- *Contraintes* : `requests` ; boucle `for` sur le dictionnaire ; recherche du maximum par comparaison (interdiction d'utiliser `max()`) ; arrondi `round(..., 2)` pour les conversions ; si le réseau est indisponible, remplacer temporairement la réponse par le dictionnaire fourni.

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

- Pourquoi demander de chercher le maximum « à la main » plutôt que d'utiliser `max()` ? Qu'apprend-on en écrivant cette boucle ?
- Comment initialiser la variable qui mémorise la meilleure devise trouvée ?
- Pourquoi comparer les taux et non les montants convertis ? (le budget est le même pour toutes les devises)
- Que représente la clé `"base"` du JSON ?

Point de validation

Vérifie ton programme : avec un budget de 100, il doit afficher `100.0 EUR = 109.0 USD`, `100.0 EUR = 85.0 GBP`, `100.0 EUR = 94.0 CHF`, `100.0 EUR = 15840.0 JPY`, puis `Taux le plus eleve : JPY ( 158.4 )`.

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

À finir à la maison

Mon convertisseur de budget de voyage.

Contexte. Tu pars en voyage dans un pays dont la monnaie n'est pas l'euro. Tu veux un programme qui convertit ton budget en devise locale, puis qui suit tes dépenses sur place et indique ce qu'il te reste, en devise locale et en euros.

Objectif. Écrire deux modules documentés — `monnaie.py` (conversion de devises) et `budget.py` (suivi des dépenses) — et un programme `voyage.py` qui :

— récupère les taux de change via l'API de démonstration du TP ;

— demande le budget en euros et la devise de destination ;

— affiche le budget converti en devise locale ;

— saisit les dépenses effectuées sur place (en devise locale) ;

— affiche le total des dépenses, le solde restant en devise locale et le total des dépenses en euros.

Questions à se poser avant de coder.

— Quelles fonctions placer dans `monnaie.py` ? Quelle est leur interface (paramètres, valeur renvoyée) ?

— Pourquoi séparer la conversion (`monnaie.py`) du suivi du budget (`budget.py`) ?

— Comment tester la conversion sans réseau ? (taux fixe de secours, ou dictionnaire de démonstration)

— Que se passe-t-il si l'URL de démonstration ne répond plus ? (réponds en une phrase)

— Comment vérifier que le programme est juste ? (penser à un cas simple : taux égal à 1, aucune dépense)

Étapes suggérées.

1. Version 1, sans réseau : module `monnaie.py` avec `vers_euro` et `depuis_euro`, programme avec un taux fixe (USD 1.09). Vérifier : 100 € → 109 dollars, puis 109 dollars → 100 €.
2. Version 2 : récupérer les taux avec `requests` (comme dans l'exercice A4) ; le programme utilise le taux de la devise demandée.
3. Version 3 : saisir les dépenses en devise locale et afficher le solde restant en devise et le total des dépenses en euros (conversion retour avec `vers_euro`).
4. Bonus : tester deux devises de destination et comparer les soldes restants.

Contraintes. Modules documentés (docstring générale + docstrings par fonction) ; import propre (jamais `from module import *`) ; arrondis à 2 décimales pour les montants ; programme testé avec les valeurs de la version 1 ; les messages peuvent être écrits sans accents (ex. `depense`, `solde`) si ton éditeur pose problème.

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