EA4 – Éléments d'algorithmique
TP n° 1 : premiers pas avec Python

Exercice 1 : interactif ou non ?

Ce premier exercice permet de découvrir différents moyens d'utiliser l'interpréteur Python.

1. Depuis un terminal, lancer l'interpréteur de Python (version 3) grâce à la commande `python3`. Dans l'interpréteur, écrire et exécuter une commande qui affiche la ligne de texte `Hello World!`. Quitter l'interpréteur en tapant `ctrl-D` (touches `Control` et `D` simultanément) ou en appelant la fonction `exit()`.
2. Créer un fichier `hello.py` contenant comme unique ligne de texte la commande précédente. Sauvegarder ce fichier et taper dans un terminal la commande `python3 hello.py`. Observer le résultat. Quelle est la différence avec `python3 -i hello.py` ?
3. Lancer à nouveau l'interpréteur Python et grâce à la commande `dir()`, afficher la liste des identificateurs connus. Taper ensuite la commande `import hello` et ensuite à nouveau `dir()`. Que constate-t-on ?
4. Rendre le fichier `hello.py` exécutable et tenter de l'exécuter. Que se passe-t-il ? Rajouter la ligne *`#!/usr/bin/env python3`* au début du fichier et réessayer.
5. Comparer les calculs `13/4` et `4/2` d'une part, et `13//4` et `4//2` d'autre part. Quelle est la différence ?
6. Calculer $\sqrt{3} + 56/9.0 \times |-1/4| + 63^2$. Pour trouver la syntaxe des opérations, on peut consulter l'aide sur les nombres entiers en tapant `help(int)` et sur la bibliothèque de fonctions mathématiques en tapant `help("math")`.
7. Dans le fichier `hello.py`, créer une fonction `affiche` qui affiche `Hello World!`. Puis tester les commandes suivantes :

   ```
   import hello
   affiche()
   hello.affiche()
   ```

   Sans quitter l'interpréteur, modifier la fonction `affiche` afin qu'elle affiche également `Bonjour le monde!`. Que se passe-t-il lorsque l'on appelle `affiche` dans l'interpréteur ? Et si l'on importe de nouveau `hello` au préalable ?

   Tester maintenant les commandes suivantes :

   ```
   import imp
   imp.reload(hello)
   hello.affiche()
   ```

   Finalement, tester les commandes suivantes dans un nouvel interpréteur :

   ```
   from hello import affiche
   affiche()
   hello.affiche()
   ```

8. Ajouter au fichier `hello.py`, une fonction `dev` qui affiche `developpeur` suivi de son propre nom.

Tester ensuite les commandes suivantes dans un nouvel interpréteur :

```
from hello import *
affiche()
hello.affiche()
dev()
hello.dev()
```

Exercice 2 : variables, expressions conditionnelles

On utilise encore l'interpréteur Python. On peut utiliser `del Z` pour supprimer la variable `Z` (la rendre non-définie) lors des tests et, grâce à la commande `dir()`, ont peut toujours vérifier si elle apparaît dans la liste des identificateurs connus.

1. Expliquer la différence entre une variable non-définie et une variable dont la valeur est `None`. Pour cela, exécuter l'instruction `print(Z)` quand `Z` est non-définie, vaut `None`, ou vaut la chaîne vide.

2. Écrire une expression qui s'évalue à `True` lorsque `Z` vaut `None` et à `False` lorsque `Z` est définie et différente de `None`.

3. Écrire une expression qui s'évalue à la chaine `"sans valeur"` si `Z` vaut `None`, `"chaine vide"` si `Z` vaut la chaîne vide et `"autre"` dans les autres cas où `Z` est définie. (Attention, il s'agit d'une expression et pas d'un segment de code qui affiche la valeur.)

4. À quelle valeur l'expression `True if x > 0 else False` s'évalue-t-elle pour `x` ayant les valeurs `3, -5, None` ou non-défini ?

Même question pour l'expression `None if x==None else True if x > 0 else False`.

5. Dans le fichier `tp1_ex2.py`, compléter la fonction `expression_5` qui retourne une expression valant `True` si $x > 0$ et `False` si $x \leqslant 0$ ou si x vaut `None`.

Exercice 3 : listes

1. En utilisant la fonction `range`, afficher les entiers de 0 à 9 (un par ligne).

2. À l'aide des fonctions `range` et `list`, créer les listes suivantes :
- la liste des entiers consécutifs de 0 à 9,
- la liste des entiers consécutifs de 2 à 10,
- la liste des entiers pairs de 2 à 10,
- la liste décroissante des entiers pairs de 10 à 2.

3. On peut également créer une liste en la définissant *en compréhension*, c'est-à-dire en décrivant comment construire ses éléments, de la façon suivante : `[<expression> for <element> in <iterable>]`

Par exemple, `[i+1 for i in range(3)]` crée la liste `[1, 2, 3]` et `[i for i in 'abc']` crée la liste `['a', 'b', 'c']`.

a. En utilisant ce mécanisme, créer les listes `L1 = [0, 2, 4, 6, 8, 10, 12]` et `L2 = ['a', 'b', 'c', 'd', 'e', 'f']`.

b. En utilisant la méthode `reverse`, inverser l'ordre des éléments de `L1`.

c. En utilisant la fonction `zip`, créer la liste
`L3 = [('a', 12), ('b', 10), ('c', 8), ('d', 6), ('e', 4), ('f', 2)]`

4. `L[a:b:p]` correspond à la sous-liste (tranche) de la liste `L` contenant les élements de `L` d'indices `a`, `a+p`, ..., `a+kp` avec `k` maximal tel que `a+kp` reste dans l'intervalle `[a,b[` : en particulier, si $\texttt{p} > 0$ et $\texttt{b} > \texttt{a}$, $\texttt{b-p} \leqslant \texttt{a+kp} < \texttt{b}$. Chaque paramètre `a`, `b` ou `p` peut être omis ; le pas `p` par défaut est 1, et les bornes sont les extrémités de la liste. Par exemple, `L[:4]` est la sous-liste formée des 4 premières cases de `L`, alors que `L[3::-1]` est son image miroir.

Ce mécanisme s'appelle *trancher* (*slicing* en anglais) parce que les valeurs de `a` et `b` correspondent aux positions des "coups de couteau" où l'on veut trancher (la position 0 correspond à l'entame avant le premier élément, la position 1 à celle entre le premier et le deuxième élément et ainsi de suite).

a. Afficher la sous-suite de `L3` contenant les éléments d'indices 2 à 4.

b. Afficher la sous-suite de `L3` contenant les éléments d'indices impairs.

c. Copier la liste `L3` dans une nouvelle liste `L4`.

d. Définir une liste `L5` contenant les 20 plus petits multiples positifs de 7. Dans la liste `L5`, mettre à 0 tous les éléments se trouvant dans une position d'indice égal à 2 modulo 3.

5. La méthode du crible d'Érathostène permet de calculer tous les nombres premiers compris entre 0 et certain n fixé. En sachant que 0 et 1 ne sont pas premiers, les entiers $2, 3 \ldots, n$ sont balayés en ordre croissant et si un entier est encore présent, alors on élimine de la liste tous ses multiples supérieurs ou égaux à son carré (parce que les autres ont été déjà éliminés lors de passages précédents).

En utilisant le mécanisme des tranches, implémenter une fonction `crible_eratosthene (n)` qui retourne la table où `cr[i]` est `True` si et seulement si i est premier.

6. Définir une liste `L` contenant les entiers égaux à 2 modulo 3 et plus petits que 36 en ordre croissant. Ensuite par le mécanisme des tranches, créer deux listes `LG` et `LD` contenant respectivement la moitié gauche et la moitié droite de `L` (de même taille à une unité près).

Créer une fonction `f` qui ajoute 2022 au premier élément d'une liste (d'entiers) passée en paramètre ; la fonction doit faire la modification sans faire aucun `return`. De même, créer une fonction `g` qui ajoute 2022 au dernier élément d'une liste.

Afficher le contenu des trois listes `L`, `LG` et `LD` après leur création ainsi qu'après chacune des opérations suivantes :
- incrémenter le premier élément de `LD` de 2022 ;
- appeler `f` sur `LG` ;
- appeler `g` sur `L` ;
- appeler `g` sur la tranche de `L` correspondant à sa moitié gauche (sans passer par une autre variable).

7. Dans le fichier `tp1_ex3.py`, compléter la fonction `somme_impairs(x)` qui calcule la somme de tous les entiers impairs de 1 à `x` en effectuant des additions.

8. Compléter la fonction `test_somme(n)` qui vérifie pour chaque entier positif `x` inférieur à `n` que la somme des nombres impairs de 1 à `x` est égale à $(\texttt{x/2})^2$ si `x` est pair et à $(\texttt{(x+1)/2})^2$ si `x` est impair.
9. Si l'on importe le fichier `tp1_ex3.py` dans l'interpréteur Python, qu'affiche l'aide en ligne `help(tp1_ex3.somme_impairs)` ? Et celle de `testDataSomme` ? Regarder le code de ce fichier pour comprendre la différence.

Exercice 4 : opérations arithmétiques revisitées

Lors du premier cours, on a vu plusieurs algorithmes pour les opérations d'addition ($n_1 + n_2$) et de multiplication ($n_1 \times n_2$) sur les entiers. Le but de cet exercice est de vous faire calculer le nombre d'opérations élémentaires utilisées par chaque algorithme, afin de comparer leur complexité. Pour cela il faudra compléter le fichier `tp1_ex4.py`.

1. Compléter la fonction `addition(nb1, nb2)` qui effectue l'addition d'entiers vue en cours, pour qu'elle retourne, en plus du tableau résultat, le nombre d'opérations arithmétiques élémentaires effectuées.
2. Compléter la fonction `additionV(nb1, nb2)` qui reprend la fonction d'addition écrite ci-dessus mais de façon à ce qu'elle puisse également s'appliquer à des tableaux de tailles différentes. Cette modification change-t-elle le nombre d'opérations élémentaires effectuées ?
3. Compléter les fonctions qui effectuent la multiplication d'entiers vue en cours pour qu'elles retournent, en plus du tableau résultat, le nombre d'opérations arithmétiques élémentaires effectuées lors dun appel à chacune d'entre elles.

   Tester ces fonctions sur de grands nombres en ajoutant des tests dans les fonctions `testDataMul`. Ces mesures reflètent-elles les complexités vues en cours ?