Structures algorithmiques et fonctions

Objectif : l'objectif de ce tp est de rendre l'élève capable de :

- gérer efficacement un ensemble de fichiers correspondant à des versions successives d'un fichier source
- manipuler un environnement de développement.
- travailler avec **Python** à travers deux interfaces **Idle** et **Pyzo**
- rechercher une information au sein d'une documentation en ligne et d'analyser des exemples fournis dans cette documentation (ici, le sujet du Tp)

1 Organisation !

1.1 Fichier Tp

Chaque Tp pourra être évalué. Il se peut que vous ayez à rendre un document permettant de suivre votre travail. Il faut donc prendre l'habitude de travailler dans un fichier nommé :

`nomp-nomTP.py` ce qui donne pour M. Gondor et ce Tp : `gondorg-INTRO-Tp-2.py`

Remarque : le nom du Tp est écrit en bas à droite du sujet.

Q - 1 : *Si l'arborescence de dossiers n'a pas été créée au Tp précédent, reprendre le script* ***Python*** *et l'exécuter pour créer l'ensemble des répertoire.*

Q - 2 : *A l'aide de la console, se placer dans le répertoire* `~/INFO/1-INTRO/INTRO-2`.

Rappel si le dossier existe, vous pouvez directement taper `cd ~//NFO/1-INTRO/INTRO-2` pour vous y rendre.

Q - 3 : *Créer (et ouvrir par la même occasion) le fichier* `nomp-nomTP.py` *en tapant :*

```
idle3 nomp-nomTP.py
```

Idle se lance est ouvre l'éditeur avec un fichier nommé `nomp-nomTP.py`.

Q - 4 : *Éditer ce fichier pour y entrer l'entête suivante :*

```
"""
Auteur : Prenom Nom
Date : 2023/09/23
Theme : IDE
"""

# Question 1 :
```

Remarque : les geeks utilisant Emacs sous linux pour taper leurs codes LaTeX, **Python** et autres, doivent rentrer les lignes suivantes :

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-
```

Lorsqu'on partage des programmes, il est important de préciser la nature de l'encodage et avec quelles versions ils doivent être compilés.

Q - 5 : *Écrire les lignes de codes ci-dessus tout en haut de votre programme.*

1.2 Sauvegardes

Il n'est pas rare en programmation d'avoir à effacer une partie du travail pour repartir sur de bonnes bases. Afin de ne pas tout perdre, il est important de conserver parfois des versions. Pour s'y retrouver on peut ajouter au nom du fichier soit :

- le numéro de version (V-1234)
- la date (2023-09-18)
- l'heure (16-33)
- un commentaire (Essai-par-dichotomie)
- une combinaison des précédentes possibilités (2023-09-18-methode-implicite)

Sous **Idle** , chaque compilation en appuyant sur `F5` vous invite à sauvegarder le fichier. Ce n'est pas forcément le cas sous **Pyzo** .

Attention ! il est préférable de ne mettre ni caractère accentué ni espace dans le nom des fichiers et des dossiers.

2 Jouons avec Python !

2.1 Commentaires, entrée/sortie et affectation : #, input, print, =

Q - 6 : *Recopier le programme suivant dans la fenêtre principale:*

```
print ( "Bonjour tout le monde !" )
# On demande maintenant le nom de l'utilisateur
nom = input ( "Ecrire votre nom :" )
print ( " Bonjour " , nom)
```

Dans l'exemple, `nom` est une variable. Elle stocke la chaîne de caractère entrée par l'utilisateur. La commande `input` permet d'entrer du contenu à partir du clavier.

L'opérateur = est l'opérateur d'affectation. Il permet d'écrire des données dans une variable. L'affectation se fait toujours de la droite vers la gauche, c'est-à-dire que la valeur à stocker doit toujours être du côté droit du signe égal alors que le nom de la variable dans laquelle on veut stocker cette valeur est toujours du côté gauche. Il est ensuite possible d'utiliser cette variable dans le programme.

Le texte derrière le croisillon # est un commentaire. Il n'est pas pris en compte par **Python** , mais permet de rendre plus clair un programme par le lecteur. On peut également librement sauter des lignes dans un programme.

Q - 7 : *Modifier le programme pour obtenir et afficher le nom et le prénom de l'utilisateur.*

2.2 Un peu d'autonomie

Remarque : pour ne pas tout compiler, il est possible de mettre en commentaire les exercices précédent. Pour cela, il suffit de sélectionner des lignes et de presser les touche `Alt+maj+3`. Deux `##` apparaissent sur la gauche. Pour faire l'opération inverse presser les touche `Alt+maj+4`.

Jouons. Faire en toute autonomie au moins un des trois jeux suivants :

Q - 8 : *Écrire un programme qui élève un nombre au cube.*

Q - 9 : *Écrire un programme qui, partant du prix hors taxe d'un article, ajoute la TVA (de 20%) et affiche* `"Le prix TTC est de..."`

Q - 10 : *Écrire un programme qui demande le prénom et l'année de naissance d'une personne, puis qui renvoie une phrase comprenant le prénom et l'âge de la personne.*

3 Structures plus élaborées

Au commencement était l'interpréteur dans lequel on tapait les instructions. C'était rapide mais pas réutilisable. L'éditeur fut donc introduit afin de conserver le programme dans un fichier, l'exécution se faisant souvent avec la touche `F5` (**Python** , **Pyzo** , **Scilab**).

Pour enrichir le programme, nous allons introduire des structures plus élaborées.

3.1 Fonctions

Une fonction est un objet qui prend des valeurs en paramètres et, après calculs, manipulations et autres, renvoie un autre objet via l'instruction `return`. Un fonction correspond donc à un ensemble d'instructions que l'on pourra *appeler* plus loin dans le programme, ce qui évitera de réécrire plusieurs fois de suite les mêmes instructions ou types d'instructions.

Une fonction s'introduit avec le mot-clé `def` et il ne faudra pas oublier le caractère `:` en fin de première ligne pour dire à **Python** qu'un nouveau bloc commence.

Exemple : fonction prenant 3 paramètres en entrée:

```
1 def ma_fonction(param1, param2, param3):
2         instruction1
3         instruction2
4         return objet_a_renvoyer
```

On peut remarquer que les lignes 2 à 4 des instructions ci-dessus sont décalées. On dit qu'elles sont indentées d'une tabulation (cf touche `tab` du clavier). L'indentation est capitale en **Python** . Un retour à la ligne signe la fin de la définition de la fonction.

Q - 11 : *Écrire les lignes suivantes dans l'éditeur et taper* `tab(3,4)` *dans l'interpréteur :*

```
# Question 11 :
def tab(a, b):
    res = a*b
    return print(a, ' * ', b, ' = ', res)
```

Comme il est saoulant d'écrire dans l'éditeur et tester dans l'interpréteur vous pouvez (devez...) **inclure** le test dans le programme. Cela permettra à votre gentil correcteur de tester très rapidement vos fonctions...

```
# Question 11 :
def tab(a, b):
    res = a * b
    return print(a, ' * ', b, ' = ', res)

# Test de la Question 11
tab(3, 4)
```

Voilà qui est pratique mais dans la vraie vie, on aime bien récupérer la sortie d'une fonction. On évitera donc les `print` dans la fonction et on les gardera uniquement pour le débogage et l'observation des résultats.

```python
# Question 11 :
def tab(a, b):
    return a * b

# Test de la Question 11
print(tab(3, 4))
```

3.2 Utilisation des bibliothèques

Nous sommes maintenant capable de créer nos fonctions. On peut aussi utiliser celles des autres pour gagner du temps ! De plus, ces fonctions sont parfois optimisées pour s'exécuter très rapidement.

Exemple : : on souhaite calculer la racine carrée ou le sinus d'un nombre. Le **Python** ne dispose pas, de base, de la fonction racine carrée. Pour pouvoir en bénéficier, il faut utiliser une bibliothèque de fonctions (que l'on appelle aussi module). Dans notre cas, nous allons importer la bibliothèque **math**.

Pour cela, il faut écrire, en tête du programme, une des lignes suivantes :

- `from math import *` : on importe alors la totalité de la bibliothèque mathématique
- `from math import sqrt, sin, pi` : on importe alors uniquement les deux fonctions souhaitées
- `import math as m` : on écrira alors `m.sqrt(3)` (resp. `m.sin(pi/4)`) pour calculer $2.\sqrt{3}$ (resp. $\sin(\pi/4)$).

La troisième méthode peut sembler un peu lourde de prime abord. Elle permet toutefois d'éviter des conflits si une même fonction est définie dans plusieurs bibliothèques. Elle permet également de disposer d'une aide contextuelle lors de la frappe ; il suffit de taper `m.` et ensuite un menu déroulant apparait avec toutes les fonctions disponibles.

Voici quelques fonctions de la bibliothèque math :

`sqrt()`	Racine carrée d'un nombre
`pow(x ,y)`	Calcule x à la puissance y
`sin()`	Sinus d'un angle (en radian)
`cos()`	Cosinus d'un angle (en radian)
`tan()`	Tangente d'un angle (toujours en radian)
`pi`	Une valeur approchée précise du nombre π

Q - 12 : *En physique, on montre que pour une faible amplitude d'oscillation, la période d'oscillation* T *d'un pendule simple de longueur l est donnée par la formule :* $T = 2.\pi.\sqrt{\frac{l}{g}}$ *avec* $g = 9,81 m.s^{-2}$. *Écrire un programme qui donne la période d'oscillation en fonction de la longueur l indiquée par l'utilisateur.*

3.3 Instructions conditionnelles

Lorsque dans un programme plusieurs cas se présentent et que des tests sont nécessaires, il est possible d'utiliser la structure suivante :

- if apparaît toujours sur la première ligne de la structure conditionnelle.
- elif apparaît si au moins un deuxième test doit avoir lieu
- else apparaît pour utiliser tous les cas non traiter par `if` et `elif`

```python
if test_1 :
    instruction_1
elif test_2:
    instruction_2
                ....
elif test_n:
                instruction_n
else:
    instruction_n+1
```

On peut alors utiliser les tests suivant :

- `x > y` renvoie `True` si `x` est strictement plus grand que `y`
- `x >= y` renvoie `True` si `x` est plus grand ou égal à `y`
- `x < y` renvoie `True` si `x` est strictement plus petit que `y`
- `x <= y` renvoie `True` si `x` est plus petit ou égal à `y`
- `x == y` renvoie `True` si `x` est égal (ou identique si on ne compare pas de simples nombres) à `y`
- `x != y` renvoie `True` si `x` est différent de `y`

Il est possible d'étendre ces tests avec les opérateurs `and` (et) et `or` (ou).

Q - 13 : *Ecire un programme `comparea(a,b)` qui retourne le plus grand nombre parmi $\{a,b\}$ et qui en cas d'égalité signale que les nombres sont égaux.*

Q - 14 : *Ecrire un programme `fac(n)` qui permet de savoir si $n \in \mathbb{N}$ est multiple de 2 et/ou 3 et/ou 4 et/ou 5.*

3.4 Boucle conditionnelle `while`

On cherche parfois à faire des opérations jusqu'à ce qu'une condition soit atteinte. On utilise alors la boucle while (tant que). Tant qu'une condition n'est pas vérifiée, on exécute plusieurs fois la même suite d'instructions.

Attention ! il faut bien réfléchir à ce qu'on fait car si la condition de sortie de boucle n'arrive jamais, le programme tourne toujours...

Q - 15 : *Tester la fonction `dectobin` de conversion d'un nombre entier `n` de la base décimale en base 2 :*

Q - 16 : *Écrire une procédure `nemequittepas` qui ne prend aucun argument en entrée. La procédure demande d'entrer une lettre. Si cette lettre est différente de `q` (pour quitter !), la lettre est imprimée et on demande la lettre suivante ; sinon on imprime `fin` et le procédure s'arrête.*

Q - 17 : *Écrire une fonction `presence` qui prend en argument un élément `x` et une liste `L`. La fonction renvoie `True` si `x` est un élément de `L` et `False` sinon.*

Q - 18 : *Écrire une fonction `indice` qui prend en argument un élément `x` et une liste `L`. La fonction renvoie l'indice de la première apparition de `x`, si `x` est un élément de `L` et `-1` sinon.*

```
# Question 15
def dectobin(n) :
    r = ""
    while n != 0:
        if n%2==0:
            r = "0" + r
        else :
            r = "1" + r
        n= n // 2
    return r

# Test de la question 15
print(dectobin(8))
print(dectobin(9))
```

Q - 19 : *Écrire une fonction `indices` qui prend en argument un élément `x` et une liste `L`. La fonction renvoie les indices des deux premières apparitions de `x`, si `x` est deux fois présent dans la liste `L` et `-1` sinon.*

3.5 Boucle inconditionnelle `for`

Si on veut répéter une instruction sur un nombre fini de valeurs, on peut utiliser une boucle `for`. On peut s'en sortir avec une boucle `while`, mais c'est parfois plus rapide avec une boucle `for`:

```
for i in range (10):
    print(i)
```

```
i=0
while i < 10:
    print(i)
    i=i+1
```

Remarque : on n'est pas obligé de prendre les n premières valeurs de $\mathbb{N}$

Instruction	Valeurs successives de i
`for i in range(2, 5)`	`2 ; 3 ; 4`
`for i in range(-10, 10, 2)`	`-10 ; -8 ; -6 ; -4 ; -2 ; 0 ; 2 ; 4 ; 6 ; 8`
`for i in range(10, 5, -1)`	`10 ; 9 ; 8 ; 7 ; 6`

Il est possible de faire tenir les instructions précédentes en une seule ligne : `[print(i) for i in range(10)]`

On parle alors de liste écrite par compréhension. On peut ainsi facilement construire des listes :

```
L = [2*i for i in range(10)]
```

pour obtenir une listes croissantes des nombres pairs.

Q - 20 : *Écrire une fonction `carre` qui prend en argument un entier `n` et renvoie la somme des nombres impairs entre 1 et $2.n-1$ (inclus), soit les n premiers nombres impairs.*

Q - 21 : *Écrire une fonction `taille` qui prend en argument un itérable `ite` et renvoie son nombre d'éléments.*

3.6 Dictionnaires

Plus fort que les listes, **Python** permet d'utiliser les dictionnaires. Au lieu d'accéder à un élément par son indice, on y accède par sa clé. La clé est non mutable mais la valeur associée à la clé est quelconque. Tout dépend de ce qu'on veut y ranger.

Pour faire un ensemble (`set`), il suffit d'écrire les éléments entre parenthèse et séparés par des virgules. Pour un dictionnaire, il faut donner pour chaque élément la clé qui lui est associée {`cle_1: val_1, cle_2: val_2`}:

```
>>> dic_sup = {"sup-1": "MPSI-1", "sup-2": "MPSI-2",
        "sup-3": "MP2I", "sup-4": "PCSI-1", "sup-5": "PCSI-2"}
>>> dic_sup["sup-5"]
'PCSI-2'
```

```
>>> dic_cub = {x:x**3 for x in range(5)}
>>> dic_cub
{0: 0, 1: 1, 2: 8, 3: 27, 4: 64}
>>> dic_cub[4]
64
```

```
>>> dic_cub[10] = 2
>>> dic_cub
{0: 0, 1: 1, 2: 8, 3: 27, 4: 64, 10: 2}
```

Q - 22 : *Construire un dictionnaire `dic_mois` pour lequel les clés sont les entiers entre `1` et `12` et les valeurs, les mois de l'année correspondants.*

On donne une liste `L` de `n` entiers compris entre 0 et `m`.

Q - 23 : *A partir de la liste `L`, construire un dictionnaire `dic_nb` dont les clés sont les entiers présents dans `L` et les valeurs correspondent au nombre d'occurrences des clés dans `L`.*