TP INFORMATIQUE 1
Découverte de Python

L'environnement de travail

Description de l'environnement Pyzo par le professeur

Python est un langage de programmation : il permet de créer toutes sortes de programmes, comme des jeux, des logiciels...
Il peut fonctionner sous les différents systèmes d'exploitation Windows, Linux et Mac OS.
Il sera possible d'associer à Python des bibliothèques afin d'étendre ses possibilités.

Pour programmer en Python, nous allons utiliser l'environnement Pyzo.On remarque qu'il y a plusieurs fenêtres dont deux principales :

- la fenêtre Console (shell) qui permet de travailler en mode interactif.
- la fenêtre Editeur qui permet de programmer et sauvegarder, programmes ensuite exécutés dans le shell

Consignes de démarrage

- Dans votre dossier personnel, créer un répertoire `Informatique` et créer un sous-répertoire `TP1` dans lequel vous pourrez sauvegarder vos fichiers Python relatifs au TP.
- Ne pas utiliser de caractères accentués ni d'espaces dans vos noms de fichier.
- Sortir un stylo, du brouillon et de quoi prendre des notes (à conserver !) sur ce que vous apprenez au fur et à mesure.

Manipulations des nombres

Dans ces premiers exercices, on réalise les opérations dans le shell.

Exercice 1. Quelques opérations courantes

1. Tester les opérations :
   - `3 + 4`
   - `-2 + 93`
   - `2 * 12`
   - `28 / 3`
   - `13,5 / 3`
2. Que font les opérations suivantes ?
   - `10 // 3`
   - `10 % 3`
   - `2**5`
3. Tester les instructions suivantes. Commenter.
   - `type(3.0)` et `type(3)`
   - `type(10/2)` et `type(10//2)`

Exercice 2. Importation de modules

1. Tester `sqrt(2)` pour calculer $\sqrt{2}$ (on utilise `sqrt` pour *square root*).
   Que se passe-t-il ?
   Explication : la fonction `sqrt` n'est pas dans python initialement. Elle est dans le module `math`.

Méthode pour charger un module.

Pour charger un module de nom *librairie*, en la renommant *lib* (pour faire plus court !), il suffit d'écrire l'instruction :

`import` *librairie* `as` *lib*

Pour utiliser la fonction *essai*, il faut alors écrire :

lib.essai(paramètres)

Exemple.

```
import math as m
a = m.sqrt(2)
```

On peut aussi importer un module sans le renommer

```
import math
a = math.sqrt(2)
```

ou importer uniquement une fonction précise

```
from math import sqrt
a = sqrt(2) # sans math
```

2. Calculer $\sqrt{2}$ en trouvant la commande.
3. Effectuer l'instruction `help(math)` pour ouvrir l'aide du module `math`.
   En utilisant cette aide, calculer e^3 et $\ln(2)$.

Stockage de données

Heureusement Python permet de stocker des données (nombres ou pas...) dans des variables. On utilise pour cela la syntaxe `ma_variable = ...`

Pour déclarer/affecter une variable `nom_de_la_variable`, le nom de la variable peut être composé de lettres, majuscules ou minuscules, de chiffres et du symbole `"_"` appelé *underscore*.

Exercice 3. Variables et affectations

1. Ranger dans la variable `mon_age` l'âge que vous aviez il y a 13 ans.
2. Écrire l'opération permettant d'actualiser votre âge, tout en conservant le même nom de variable.
3. Lire les messages d'erreur renvoyés par l'interpréteur quand vous écrivez :
   - `mon-age = 18`
   - `2013_mon_age = 18`
   - `True = 18`

   Quels sont les problèmes rencontrés ?
4. Ecrire `age = 5`, valider puis écrire `age = Age + 14`. Valider et noter le message d'erreur. Quel est le problème ?

Exercice 4. Variables multiples

Python permet la gestion de couples de variables, de triplets ou plus généralement de « *tuples* »

1. Tester successivement dans le shell les commandes suivantes :

```
>>> a=12
>>> b=78
>>> (a,b)
>>> (c,d) = (a,b)

>>> c
>>> d
```

2. On peut aussi stocker des tuples dans de nouvelles variables.
   Tester les commandes suivantes :

```
>>> a=12
>>> b=78
>>> u = (a,b)
>>> type(u)

>>> (c,d) = u
>>> c
>>> d
```

3. Dans l'instruction précédente `(c,d) = u` on dit qu'on a « dépaqueté u » (unpack en anglais)
   Tester les commandes suivantes.

```
>>> a=12
>>> b=78
>>> u = (a,b)
>>> len(u)

>>> (c,d,e) = u # expliquez le message d'erreur
>>> (a, a) = u # que contient a ? Vérifiez

>>> v = ( u, (4,2) )
>>> v # Combien d'éléments contient v ?
#Vérifiez avec len(v)
```

Modularité de la programmation

Votre code se doit d'être organisé et d'éviter les redondances. Nous allons voir comment procéder.

Exercice 5. Scripts - organisation du code

1. Fermer l'environnement Pyzo. Relancer. Vos instructions ont-elles été conservées ?
2. Dans la barre d'outils, cliquer sur "Fichier" puis sur "Nouveau" pour ouvrir l'éditeur. Enregistrer le fichier sous le nom `premier_script` dans le répertoire $TP1$.
   Taper dans ce fichier :

```
a = 12 * 12
b = 3.14
c = 42

moy = (a+b+c)/3
```

3. *Sauvegarder puis exécuter ce script.*
Tel quel ce script n'affiche rien mais les variables ont bien été créées.
Vérifiez-le en les appelant dans le shell.
Vous pouvez même si vous le souhaitez, importer ce script comme n'importe quel module python avec la commande **import** `premier_script` dans d'autres scripts python.
Il suffit de placer le fichier `premier_script.py` dans le dossier où se situent les autres scripts.

N.B. Dans une utilisation avancée, c'est même une bonne pratique de placer les fonctions dans un script différent du script principal (celui où vous utilisez vos fonctions). Mais bon, nous n'en sommes pas encore là...

Dorénavant, vos algorithmes seront écrits dans l'éditeur de scripts. Le shell sera réservé aux tests.

Remarque. Insérer des `print` dans le script pour afficher vos variables est à proscrire !
Cela peut rapidement provoquer une multitude d'affichages parasites.
Pour nous le `print` servira essentiellement à débuguer un script en affichant temporairement certaines variables clés (on les enlève ensuite dans ce cas) ou bien à déclencher des affichages voulus.

Exercice 6. Fonctions - un pas vers l'interactivité

1. Pour l'instant notre seule manière d'interagir avec le script est de le rouvrir, modifier les valeurs de a, b, c puis l'exécuter à nouveau.
Modifiez le script comme suit pour créer la fonction `moyenne`.

```
def moyenne(a,b,c):
    """moyenne arithmétique des nombres a,b,c """
    moy = (a+b+c)/3
    return moy
```

Bien noter les indentations en début de ligne à l'intérieur de la fonction.

Si vous n'oubliez pas les deux point à la fin de la première ligne, l'indentation se crée automatiquement en appuyant sur Entrée

Après exécution du script la fonction `moyenne` est directement utilisable.
Testez la dans le shell.
Remarque. Il est possible d'imposer des contraintes aux paramètres en entrée avec l'instruction `assert`.
Tester la fonction suivante avec différentes valeurs de x. Qu'obtient-on en enlevant le `assert` *?*

```
def racine_car(x):
    """racine carrée d'un nombre positif x"""
    assert  x >= 0
    return x**0.5
```

2. Vous pouvez dans le même script définir plusieurs fonctions qui s'appellent mutuellement.
Dans la suite on modélise points et vecteurs du plan par des couples (leurs coordonnées).
Par exemple le vecteur $\vec{i}$ correspondra au couple `(1,0)` en python. Le point $A(1,3)$ correspondra au couple `(1,3)`.
Écrire successivement (chaque fonction utilisera les précédentes)

- une fonction `norme(u)` qui renvoie la norme du vecteur `u`
  On rappelle qu'on peut dépaqueter un couple avec la commande `(a,b) = u`
- Une fonction `vecteur(A, B)` qui retourne le couple des coordonnées du vecteur $\overrightarrow{AB}$.
- Une fonction `distance(A,B)` qui retourne la distance entre les points A et B (la norme du vecteur $\overrightarrow{AB}$).

Structures de contrôle

Le but d'un algorithme est d'automatiser certaines actions :
- exécution d'actions différentes suivant certaines conditions,
- répétition d'actions suivant certains paramètres / certaines conditions.

Exercice 7. Booléens et test conditionnel

1. Tester les instructions suivantes.

```
>>> a = 13
>>> a
>>> a == 13
>>> a == 15
>>> type(a == 15)
```

Que représente le symbole `==` ?

À l'aide de tests, déterminer ce que signifient les opérateurs `<`, `<=`et `!=` sur des nombres.

2. Recopier et tester les fonctions suivantes. Au passage examiner la syntaxe des tests.

```
def est_carre(x):
    if x >=0:
        return True
    else:
        return False

def est_cos(x):
    if x >1:
        return False
    elif x<-1:
        return False
    else:
        return True
```

A l'aide des connecteurs **and** / **or**, proposer une modification de la fonction `est_cos` qui évite l'imbrication de tests.

3. Un grand classique.

Pour a, b, c des paramètres réels, écrire une fonction `racines(a,b,c)` qui retourne les racines du l'équation du second degré $ax^2 + bx + c = 0$.

Remarques. Vous aurez à retourner des *couples* de racines le cas échéant.

Dans le cas de racines complexes, chaque racine $u + iv$ sera elle-même représentée sous la forme d'un couple `(u,v)`.

Pensez à faire en sorte que votre fonction vérifie si l'équation est vraiment du second degré.

Exercice 8. Boucle for

1. Tester la fonction mystère suivante

```
def mystere(n):
        """ documentation sur la fonction """
        s = 0
        for i in range(1,n):
                print(i) # pour comprendre
                s = s + i
        return(s)
```

2. Quelles valeurs prend la variable i au cours de l'exécution.
Comparer avec les paramètres de **range**.
Dès que vous aurez compris comment évolue i, supprimez l'instruction **print(i)** *qui pique les yeux.*

3. Que calcule cette fonction ?

4. Adapter cette fonction pour calculer la factorielle d'un entier.

Exercice 9. Boucle while

La boucle for précédente répète une action pour certaines valeurs précises d'un paramètre géré de manière automatique par python.

En particulier le nombre de « tours de boucle » est déterminé à l'avance.

La boucle while intervient lorsqu'on a seulement une condition pour que la boucle continue.

1. Tester la fonction suivante de paramètre un nombre `seuil`.
On pourra en particulier tester pour les valeurs 16 et 17

```
def boule_de_gomme(seuil):
    n = 0
    while 2**n < seuil :
        n = n+1
    return n
```

Que calcule-t-elle ?

2. Écrire une fonction `depasse(p)` qui détermine le plus petit entier $n \geqslant 2$ tel que $n! > n^p$.
Pour $p = 42$ elle doit retourner 55.

Types de données structurés

Nous aurons très souvent besoin de stocker plusieurs données en un seul objet python.
coordonnées, valeurs successives prises par une suite, manipulation de mots/phrases, nombre d'apparitions d'une série de valeur déterminée dans une liste de nombres,...
Nous avons déjà évoqué les tuples mais d'autres objets existent ; chacun avec ses caractéristiques propres.

Exercice 10. Listes - manipulations élémentaires

On dispose en `Python` d'un type de données appelé `list`.
Il s'agit d'une suite de valeurs stockées dans des cases mémoires consécutives :
on peut le représenter ainsi

2	5	3	-1	7	2	1

- Pour déclarer cette liste en `Python`, on écrit `v = [2, 5, 3, -1, 7, 2, 1]`.
- La fonction `len` calcule sa longueur n : `n=len(v)`.
- On accède au contenu de la case numéro i de `v` par `v[i]`.

1. Déclarer la liste `v` définie ci-dessus.
Vérifier le type de votre variable. Affecter la longueur n de `v`.

2. Tester `v[0]`, `v[2]`, `v[n]`, `v[n-1]`, `v[-1]`,`v[-2]`. Commenter.
Changer la valeur du quatrième élément par 0.

3. Que renvoie `v[1:3]` ?
   On parle de « tranche » d'une liste (slice en anglais)
   Remplacer dans `v` les trois derniers éléments par des 42 en une seule affectation (à l'aide d'une tranche).
   Que fait `v[1]=[0,0,0]` ? Combien y a-t-il maintenant d'éléments dans `v` ?
4. Exécuter la commande `T = v + 2 * [1,2,3]`. Que vaut `T` ?
   (on parle de « concaténation » de listes)
   Essayer de deviner ce que contient `L` après la commande `L = [0]*10` et après `L = []*10`. Vérifiez.

Exercice 11. Listes - création itérative et copie

1. **Création de liste par compréhension**
   a. Tester l'instruction `L = [i**2 for i in range(26)]`
   b. Adapter pour construire la liste des entiers impairs compris en 5 et 601
2. **Création de liste par append successifs**
   a. Tester les commandes suivantes

```
L = [7, 3, -9, 15]
L.append(42)
```

   Qu'est devenue `L` ?
   N.B. constatez qu'on n'a pas eu besoin d'affecter à nouveau `L`.
   Que donne la commande `L = L.append(42)` ?
   b. Soit u la suite définie par $\begin{cases} u_0 = 1 \\ u_{n+1} = 1 + u_n^2 \end{cases}$.
   Complétez les lignes 4 et 7 du code suivant pour que la fonction soit correcte.

```
1  def liste_valeurs_u(n):
2      """retourne les n premiers termes de la suite u"""
3      u=1 #initialisation
4      L =
5      for i in range(1,n):
6          u = 1 + u**2 #u contient u_i
7          L
8      return L
```

3. **Copie de listes**
   a. Déclarer, en compréhension, la liste `v` des entiers pairs de 0 à 12.
   b. Exécuter les instructions suivantes

```
L = v
L[2] = 37
```

   Afficher ensuite `v` et `L`. Que constatez-vous ?
   c. Refaire le test en utilisant `L = v.copy()` au lieu de `L = v`
   d. Exécuter maintenant les commandes suivantes.

```
v = [ [13,-7] , 5 ]
L = v.copy()
v[0][0] = 42
```

   Afficher `v` et `L`.

Exercice 12. Dictionnaires

Une liste peut se visualiser comme une succession de cases numérotées dans lesquelles sont stockées des données.
Lorsque l'ordre des données est indifférent on peut utiliser les *dictionnaires*.
Ceux-ci permettent en outre d'utiliser n'importe quel nombre ou chaine de caractères comme « clé » d'une donnée.

1. Définir la variable suivante

   ```
   inventaire = { 'assiettes':24 , 'fourchettes':12 , 'verres':6 }
   ```

   Vérifier son type et sa longueur en python.

2. On peut accéder à une entrée d'un dictionnaire avec la syntaxe `dico[clé]` et en créer une nouvelle avec `dico[clé] = valeur`.

   Insérer quelques nouvelles données dans votre inventaire.

   On peut vérifier la présence d'une clé avec l'instruction `clé in dico`.

   Faire quelques tests.

3. Soyez vigilant sur le choix des clés. *Quelle est l'erreur dans les instructions suivantes ?*

   ```
   # je rajoute mon nombre de poêles
   inventaire['poêles'] = 2

   # je le modifie car j'en ai acheté une nouvelle
   inventaire['poeles'] = 3
   ```

   Définir et afficher le dictionnaire suivant

   ```
   dico = {1:42} #initialisation
   ```

   Tester ce code

   ```
   for k in range(0,7):
       dico[1+1000**(-k) ] = k
   ```

   Afficher de nouveau `dico`. *Où est passé le 42 ?*

4. La copie des dictionnaires nécessite les mêmes précautions que pour les listes. *Faire quelques tests en vous inspirant de l'exercice précédent pour faire apparaïtre les difficultés.*

Exercice 13. Tuples et Chaines de caractères

1. **Définition de chaines de caractères**

   ```
   >>> s = 'Bonjour'
   >>> type(s)
   # str abréviation de 'string'
   # terme anglais pour 'chaine de caractères'

   # Comparez les deux instructions suivantes
   >>> t = ' Aujourd'hui '
   >>> t = " Aujourd'hui "
   ```

   On peut aussi utiliser des triple-guillemets `"""ma phrase """` *qui autorise apostrophes, guillemets, retours à la ligne, etc.*

2. **Accès aux élements**

   On définit le tuple et la chaine suivants

```
>>> t = (5, 12, -63, 3.14)
>>> s = " La physique ?\n C'est chic !"
# Tester print(s). Que représente \n ?
```

En vous inspirant des listes, trouvez les commandes permettant d'accéder individuellement aux élements/caractères de t et s.

Testez aussi les commandes de tranchages et de concaténation (avec `+` et `*`)

Que se passe-t-il si vous essayez de modifier un tuple ou une chaine ?

On dit que les tuples et les chaines de caractères sont *non-mutables* ou *immuables*.

.

Pour aller plus loin

PARCOURS D'UNE STRUCTURE DE DONNÉES

- On peut parcourir tuple, liste et chaine de caractère grâce à une simple boucle utilisant un indice

```
n = len( struct )
# struct est votre structure de données
for k in range(n):
    #instructions utilisant
    struct[k]
```

Si l'indice n'est pas utile, on peut aussi directement parcourir les éléments :

```
for elt in struct :
    #instructions utilisant
    elt
```

- Pour un dictionnaire on aura besoin des clés utilisées

```
for key in dico.keys() :
    instructions utilisant
    dico[key]
```

Exercice 14. Fonctions simples sur des données structurées

1. Écrire une fonction `somme(L)` calculant la somme des valeurs d'une liste de nombres.
2. Écrire une fonction `nb_voyelles(s)` qui compte le nombre de voyelles d'une chaine de caractères `s`.
3. On se donne un dictionnaire de la forme `dico = { val1:nb1 , val2:nb2 , ...}` où `nb` est le nombre d'apparitions de la valeur `val`.
   Écrire une fonction `somme(dico)`qui calcule la somme totale des valeurs (en tenant compte de leur nombre d'apparition).

Exercice 15. Méli-mélo de données

1. **Transformation de chaines en listes de caractères.**
   Écrire une fonction `transfo_liste(s)` qui retourne la liste des caractères utilisés dans une chaine `s`.
   Au moins deux versions possibles suivant la méthode de création de liste choisie !
2. **Séparation avancée d'une chaine**
   a. Testez la commande suivante

```
>>> phrase = 'La réponse universelle est 42.'
>>> phrase.split()
```

   b. On se donne la chaine `data = '12 ; 0.34 ; 5.6 ; -78'`
   En utilisant **help**`(data.split)` trouvez l'instruction permettant de décomposer la chaine suivant le caractère `;` .
   Dans la liste obtenue, quelle est le type de chaque élément ? Que donnerait donc l'opérateur `+` sur ces éléments ?
   c. Écrire une fonction `somme(data)` qui, à partir d'une variable `data` quelconque sous le format précédent (chaine de caractères de nombres séparés par des points-virgules), retourne la somme des valeurs correspondantes.
   (on utilisera **float**`()` *pour convertir en flottants les "valeurs" rencontrées)*