2- Séquences et Boucles

Ce que l'on conçoit bien s'énonce clairement, Et les mots pour le dire arrivent aisément.

Nicolas Boileau (l'Art poétique)

Objectifs :

- Séquences str, tuple, list ;
- Structures de contrôle : boucles for et while ;
- Rupture de boucles break et continue ;

Fichiers à produire : `TP2_1.py` `TP2_2.py` `TP2_3.py` `TP2_4.py` `TP2_5.py` `TP2_6.py` `TP2_7.py`

2.1 - Listes et tuples

Ce sont deux conteneurs, qui permettent de stocker des *séquences* de données de tout type (là où les chaînes ne permettent de stocker que des caractères), dans un ordre connu et avec la possibilité d'avoir une répétition de certaines valeurs.

Les **tuples** sont utilisés de façon quasi transparente par Python, ils permettent de créer des séquences de données que l'on ne peut plus modifier ("immutables"). On les écrit simplement entre parenthèses : `( 1, 4, "toto")`

Il est même possible, lorsqu'il n'y a pas d'ambiguïté, d'ôter les parenthèses du tuple. A noter le tuple d'un seul élément : `("un",)` qui doit contenir une virgule pour le distinguer de simples parenthèses de regroupement d'une expression calculée.

Les **listes**, qui seront très utilisées lors des TP, sont des séquences modifiables ("mutables"). On les écrit entre crochets : `[ 7, True, "Hello" ]`

2.1.1 - Indexation de séquences

Essayez les indexations sur chaînes de caractères et commentez si besoin :

```
>>> s = "Une séquence de caractères"
>>> s[5]
>>> s[-1]
>>> s[4:12]
>>> s[13:]
>>> s[:3]
>>> s[:]
>>> s[::-1]
>>> s[16] = 'C' # On essaie de mettre une majuscule
```

Vérifiez que cela fonctionne aussi sur les listes et les tuples :

```
>>> lst = [ "zero","one","two","three","four", "five","six" ]
```

```
>>> lst[2]
>>> lst[:2]
>>> lst[4:]
>>> lst[-1]
...
>>> refs = ( "zéro","un","deux",3,"quatre")
>>> refs[1:4]
...
```

Modification de séquence par l'indexation

Rappel, les chaînes de caractères et les tuples sont des séquences **immutables**, une fois créés il n'est pas possible de les modifier. Essayez :

```
>>> refs[3] = "trois"
>>> s[4:12] = "chaîne"
```

Par contre, les listes sont des séquences **mutables**, il est donc possible de modifier les valeurs en place. Ceci utilise la même notation d'indexation, mais en partie gauche d'une affectation. Essayez et commentez (ré-affichez `lst` après chaque modification pour comprendre ce qui s'est passé) :

```
>>> lst
>>> lst[1] = "un"
>>> lst[3:6] = [ 4,5,6 ]
>>> lst[2:2] = [ "inséré","plusieurs","valeurs" ]
>>> lst[-1] = "données"
>>> lst[-2] = ["un","élément","séquence"]
```

On peut aussi utiliser une notation d'indexation comme argument d'une instruction **del** pour supprimer un ou plusieurs éléments. Essayez et ré-affichez **lst** :

```
>>> dell st[5]
```

Séquences imbriquées

Il est possible avec les tuples et les listes d'utiliser comme valeur d'élément tout type de données, entre autres des séquences. Essayez :

```
>>> data=["Pierrot",(4,8,12),[(1.2,5.2),(0.1,9.3)],["un texte" ]]
>>> data[1][0]
>>> data[2][1]
```

Écrivez (exercice à mettre au compte-rendu) les instructions d'indexation imbriquées permettant de récupérer à partir de **data** les valeurs suivantes :

- *1.2*
- *"Pierrot"*
- *"rot"*
- *"texte"*

Affectation et références

Les boucles

Le parcours de séquence (for)

L'instruction **for** est l'instruction idéale pour parcourir les éléments d'une séquence, par exemple d'une liste (exécutez l'exemple suivant **TP2_for.py**) :

```
donnees = [ 1, 3.14, "Bonjour", (7,8,9), [(1,2),(3,4),(5,3)] ]
for v in donnees : t = type(v)
d = v * 2
print("Valeur: v =",repr(v),"\n\ttype de v:",t,
"\n\tdoublé de v:",d)
```

L'exécution vous permet de suivre les différentes valeurs par lesquelles est passé tour à tour **v**, son type ainsi que le résultat d'une multiplication par 2 de la valeur.

La deuxième partie des instructions dans l'exemple (**TP2_for.py**) permet de parcourir les index des

données dans la séquence et, dans le cas d'une liste (mutable), de modifier les données si l'on veut :

```
for index in
range(len(donnees)) : v =
donnees[index]
print("Valeur: v
=",repr(v))
donnees[index] = str(v)+" à l'index
"+str(index) print(donnees)
```

Séries d'entiers (`range`)

La fonction générateur **`range`** permet de créer des séries d'entiers et de les parcourir dans des boucles **`for`**.

Essayez :

```
>>> for i in range(10) :print(i)
>>> for i in range(2, 10) :print(i)
>>> for i in range(2, 11, 2) : print(i)
```

Si l'on veut avoir d'un coup la liste complète des valeurs d'un range, il est possible d'écrire :

```
>>> list(range(10))
```

Boucles imbriquées

Il est aussi possible d'imbriquer des boucles, essayez et commentez l'exemple suivant (exécutez le fichier **`TP2_forimb.py`**) :

```
print("",end="")
for col in range(1,4) :print(col,end="      ")
print()
for ligne in range(1,6) :
print(ligne,end=" : ")
for col in range(1,4) :
print(ligne*col,end="                    ") print()
```

La répétition en boucle (`while`)

L'instruction de boucle **`while`** permet de contrôler l'exécution du bloc d'instructions de la boucle ***tant que*** la condition exprimée est ***vraie***. Voyez l'exemple suivant (ouvrez et exécutez le fichier **`TP2_while.py`**), modifiez le afin de tester d'autres possibilités (changement de condition de boucle, changement de l'expression sur **`i`**, valeur de départ de **`i`** différente…):

```
i = 7
while i > 0 :                          # ne pas oublier le
deux-points ! print (i,' ',end=' ')    # ne pas oublier
l'indentation ! i = i - 1              # décrémentation
```

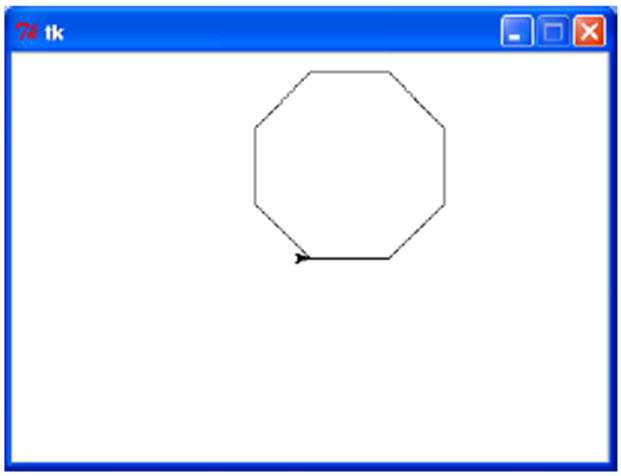
Exemple graphique

Le module ***turtle*** permet d'illustrer les boucles de façon ludique. Il offre la possibilité de réaliser des « graphiques tortue », c'est-à-dire des dessins géométriques correspondant à la piste laissée derrière elle par une petite « tortue » virtuelle, dont nous contrôlons les déplacements relatifs sur l'écran de l'ordinateur à l'aide d'instructions simples : **forward(*n*)** pour avancer de ***n*** pixels, **left(*a*)** pour tourner à gauche de ***a*** degrés, **right(*a*)** pour tourner à droite de ***a*** degrés, **reset()** pour effacer l'écran et remettre la tortue au centre…[14].

Exécutez le fichier **TP2_turtle.py**, qui contient les instructions ci-dessous ; et essayez différentes combinaisons de valeurs pour différents résultats :

```
from turtle import *
for i in range(4) :
forward(100)
left(90)
input("Appuyez sur Entrée
pour la suite")

reset
() a
= 0
while a < 12
:
forward(150)
left(150)
a = a+1
```



```
input("Appuyez sur Entrée pour la suite")
```

Programmation

2.2.1 - Multiples de 7

Programme TP2_1.py

- Afficher les 20 premiers termes de la table de multiplication par 7, en signalant au passage (à l'aide d'une astérisque) ceux qui sont des multiples de 3.

Exemple : **`7 14 21 * 28 35 42 * 49`**…

Vous vous rappellerez le paramètre **`end="xxx"`** de la fonction **`print`**, ainsi que l'opérateur modulo calculant le reste de la division entière.

Octogone

Programme TP2_2.py

- Tracer, à l'aide d'une boucle **`while`** (puis, dans un second temps, à l'aide d'une boucle **`for`**), un *octogone* de 50 pixels de côté .

Vous penserez à calculer le nombre d'itérations nécessaires, et l'angle de rotation entre le traçage de deux côtés consécutifs.

Combinaisons de dés

Programme TP2_3.py

Vous jouez avec deux dés et vous voulez savoir combien il y a de façons de faire un certain nombre. Bien noter qu'il y a deux façons d'obtenir 3 : 1-2 et 2-1.

- Saisir une valeur entière (indiquer à l'utilisateur que le nombre attendu est entre 2 et 12).
- Calculer, à l'aide de boucles imbriquées, le nombre de possibilités que deux dés à six faces produisent cette valeur.
- Afficher ce résultat.

À l'exécution l'écran devrait ressembler à :

```
Entrez un entier [2..12]. : 9

Il y a 4 façon(s) de faire 9 avec deux dés.
```

2.2.2 Somme des 10 premiers entiers

Programme TP2_4.py

- Afficher la somme des 10 premiers entiers de trois façons différentes :
  - en utilisant la formule classique : ;
  - en utilisant une boucle **`while`** ;
  - en utilisant une boucle **`for`**.

2.2.3 - Epsilon machine

Programme TP2_5.py

Un ordinateur n'est qu'un automate programmé *fini*. Il est sûr que l'on ne peut espérer effectuer des calculs avec une précision infinie, ce qui conduit à des « erreurs d'arrondi ». Quelle précision peut-on espérer ?

On va définir l'*epsilon-machine* comme la plus grande valeur ε telle que[15] :

$1+\varepsilon=1$

- Initialiser une variable `dx` à la valeur `1.0`.
- Dans une boucle `while`, diviser `dx` par `2` tant que la condition (`1.0 + dx > 1.0`) est vraie.
- Afficher la valeur dx finale. Comparer cette valeur avec une valeur normalement nulle (exemple la valeur de `sin(π)`).

2.2.4 - Liste de points

Programme TP2_6.py

On dispose d'une liste de coordonnées de points x,y sous la forme d"une liste de tuples :

```
coords = [ (4,5), (9,3), (12,8), (13,7), (18,6), (20,9) ]
```

- Créer une liste vide d'ordonnées `ordos`.
- Remplir la liste d'ordonnées en parcourant la liste des coordonnées.
- Afficher la liste d'ordonnées.

On veut maintenant filtrer le contenu de `coords` afin de borner à 7 toutes les ordonnées qui y sont supérieures.

- Parcourir la liste de `coords`, et y modifier tous les points d'ordonnée `>7` de façon à ce que pour ces points, les ordonnées soient fixées à 7.

Après votre traitement, affichez `coords`, qui devrait contenir :

```
[ (4,5), (9,3), (12,7), (13,7), (18,6), (20,7) ]
```

Enquête criminelle

Programme TP2_7.py

Vous êtes tranquille dans votre laboratoire lorsque votre responsable arrive avec plusieurs listes de personnes. Trois assassinats similaires ont eu lieu consécutivement dans trois salles de spectacle différentes. Les enquêteurs sur le terrain ont relevé les listes des présents dans chaque salle. Afin de restreindre la liste des suspects, votre première tâche est d'identifier la ou les personnes qui se sont trouvées présentes aux trois spectacles.

Votre programme commencera par une instruction :

```
from enquete_m import *
```

Qui importe dans votre programme les variables globales suivantes :

`salle1` - nom de la première salle de spectacle.

`spectateurs1` - liste des noms des personnes présentes dans la première salle.

`salle2` - nom de la seconde salle de spectacle.

`spectateurs2` - liste des noms des personnes présentes dans la seconde salle.

`salle3` - nom de la troisième salle de spectacle.

`spectateurs3` - liste des noms des personnes présentes dans la troisième salle.

Vous devez :

- A l'aide de boucles, remplir une nouvelle liste `pres12` qui soit constituée des noms des personnes présentes dans les deux premières salles.
- A l'aide de boucles, prendre ensuite en compte la troisième salle et créer une liste `pres123` qui soit constituée des noms des personnes présentes dans les trois salles.
- Vérifier que le traitement fonctionne, via un contrôle utilisant les opérations sur les ensembles, avec l'expression suivante :

```
set(spectateurs1)&set(spectateurs2)&set(spectateurs3)==set(pres123)
```

Votre résultat a permis d'interpeller un suspect. Les enquêteurs ont analysé ses faits et gestes la veille d'un des crimes, et ils ont un doute sur ce qui concerne ses déplacements. Ils vous demandent de calculer la longueur totale d'un trajet en plusieurs étapes - dont ils ont les coordonnées - et de donner les estimations de temps nécessaires au parcours à différentes vitesses.

L'import issu de `enquete_m` a aussi fourni dans votre programme une autre variable globale :

`trajet` - liste de tuples de coordonnées flottantes x,y en km,km. Par exemple :

```
[(91.68, 53.10), (81.52, 50.14), (43.68, 18.18)]
```

Vous devez :

- A l'aide de boucles, remplir une nouvelle liste `distances`, constituée des distances entre

chaque coordonnées consécutives du trajet.

- Afficher cette liste.
- Calculer dans une variable `parcouru` la distance totale parcourue en sommant ces distances, puis afficher cette variable.
- Afficher avec une présentation correcte, pour les vitesses de 25km/h, 40km/h, 50km/h, 60km/h, 75km/h, 90km/h, 120km/h, 130km/h, le temps qui aurais été nécessaire pour parcourir la distance totale. Éviter les répétitions inutile de code