LE LANGAGE PYTHON

Commentaires, importations et fonctions

```
"""
    Ceci est un commentaire
    sur plusieurs lignes
"""
# et ceci un commentaire sur une seule ligne

# DEFINIR UNE FONCTION
def add(a, b):
    c = a + b  # affectation de a + b à c
    return c   # retourne la valeur de c

# IMPORTER UN MODULE
import random # importe le module pour générer des nombres aléatoires
alea = random.randint(1,100) # doit être précédé du nom du module
from math import * # importe toutes les fonctions du module mathématiques
racine = sqrt(2) # inutile de mentionner math
```

Structures de contrôle

```
# CHOISIR
if (a == b) and (a < 1): # si a = b ET a < 1
    pass # indique à Python de ne rien faire
elif a != c or not(a == 1): # si a différent de c OU a NON égal à 1
    pass
else:
    ... # bloc de code exécuté par défaut

# PARCOURIR
for i in range(1, 6): # pour i allant de 1 à 5
    print(i) # affiche 1, 2, 3, 4, 5 en sautant des lignes
for l in "bonjour": # pour chaque lettre l du mot bonjour
    print(l, end = '') # ne saute pas de ligne après le print

# RÉPÉTER
while u <= M: # tant que u est inférieur ou égal à M
    ...
```

Type des variables

```
n = 10       # n est de type entier (int)
x = 1.0      # x est de type réel (float)
s = "m"      # s est de type string (str)
b = True     # ou False, b est un booléen (bool)
type(n)      # renvoit le type de la variable n
```

```
str(n)       # convertit l'entier n en une chaîne de caractères
bin(n)       # renvoit la représentation de n en binaire
ord('a')     # renvoit la valeur unicode du caractère
chr(97)      # renvoit le caractère défini par sa valeur unicode
s = s + "ot" # s contient la chaîne "mot"
n = n + 1    # n = 11
n = n ** 2   # élève n au carré
x = 1 / 3    # x = 0.3333333333333333
q = a // b   # affecte à q le quotient de la division euclidienne de a par b
r = a % b    # affecte à r le reste de la division euclidienne de a par b
```

Entrées/Sorties

```
# Faire saisir une valeur par l'utilisateur
s = input("s = ") # demande la valeur de s (attention, s est de type str)
n = int(s)        # convertit s en un entier
x = float(s)      # convertit s en un réel

# Affichage simple
print("n = ", n)       # affiche "n = " puis la valeur de l'entier n
print("n = " + str(n)) # autre méthode pour afficher l'entier n
print("pi = {:.2f}".format(3.14159)) # affiche 2 chiffres après la virgule

# Affichage enrichi
print("{:-^20}".format(" ALIGNEMENT ")) # ---- ALIGNEMENT ----
print("{:<20}".format("à gauche"))      # à gauche
print("{:^20}".format("centré"))        #        centré
print("{:>20}".format("à droite"))      #              à droite
print("{:-^20}".format(""))             # --------------------
```

Structures de données

```
# Type TUPLE (non modifiable)
rgb = (100, 200, 10)
red = rgb[0]             # red vaut 100
# Type LISTE (modifiable)
l = []                   # l est une liste vide
l = [1, "a", True]       # liste d'éléments de types différents
t = list(l)              # t est une copie distincte de l
len(l)                   # renvoie la longueur d'une liste, ici 3
l.append("mot")          # ajoute l'élément "mot" à la liste
l.remove("a")            # supprime "a" de la liste
n = l.pop(i)             # retire et renvoie l'élément d'indice i de la liste
n = l.pop()              # retire et renvoie le dernier élément
k = l[0]                 # affecte à k l'élément de rang 0
k = l[-1]                # affecte à k la valeur du dernier élément de la liste
l[1] = "b"               # remplace l'élément d'indice 1
## LISTE en compréhension
```

```python
liste_carres = [x**2 for x in range(5)] # liste_carres = [0, 1, 4, 9, 16]

## PARCOURS d'une liste
for i in range(len(liste_carres)): # par INDEX
    print(liste_carres[i])
for x in liste_carres:             # par VALEUR
    print(x)
for index, valeur in enumerate(liste_carres): # par énumération index/valeur
    print(index, valeur)

# Type DICTIONNAIRE (modifiable)
dico = {} # ou dico = dict() pour un dictionnaire vide
dico = {"Alice" : 28, "Carol" : 25}
dico["Bob"] = 20 # ajout d'un couple clé/valeur
del(dico["Alice"]) # suppression d'un couple clé/valeur

## PARCOURS d'un dictionnaire par couple clef/valeur
for clef, valeur in dico.items():
    print(clef, valeur)
```

Programmation objet

```python
class MyClass:
    # Constructeur de la classe
    def __init__(self, n): # toujours self en 1er paramètre
        self.x = n # x est un ATTRIBUT de classe
    # METHODE de classe
    def foo(self):
        pass
    # Surcharge de l'opérateur d'affichage appelé lors d'un print
    def __str__(self):
        return "Hello world !!!"
    # Surcharge possible des opérateurs arithmétiques +, -, *, /
    def __add__(self) # sub, mul, div
        ...

o = MyClass(10) # création d'un objet de la classe MyClass avec o.x = 10
o.foo()
print(o) # appel de __str__(self)
```

Mise en oeuvre de programme

```
# Pour répondre à une problématique, on cherchera à suivre les étapes suivantes
1 - Parcourir la structure de données par index ou par valeur avec un for
2 - Récupérer la donnée en la stockant dans une variable
3 - Appliquer un traitement à la donnée récupérée
```

COMMANDES LINUX

Général

```
clear               # efface l'écran
man cmd             # affiche le manuel de la commande -> q pour quitter
echo "texte"        # affiche le texte
echo "texte" > fic  # crée un fichier fic contenant le texte
echo "texte" >> fic # Ajoute le texte à la fin du fichier fic
```

Naviguer et s'informer

```
# Arborescence des dossiers
/           # racine du système
/home/user  # répertoire de l'utilisateur
./          # répertoire courant
../         # répertoire au-dessus du répertoire courant
# Commandes
pwd         # affiche le dossier courant
cd dossier  # se positionne dans le dossier spécifié
ls          # liste les fichiers du dossier courant
ls dossier  # liste les fichiers du dossier spoécifié
ls -al      # plus de détails sur chaque fichier
du -h fic   # affiche l'espace disque en Ko occupé par fic
ps aux      # affiche la liste des processus actifs
kill -9 PID # force l'arrêt du processus PID
```

Créer et modifier

```
touch nom_fic              # crée un fichier vide
chmod (ugoa)(+-)(rwx) fic # modifie les droits du fichier
mkdir nom_rep              # crée le dossier nom_rep
cp src dest                # copie src vers dest
mv src dest                # -renomme src en dest
rm fic                     # efface le fichier fic
rm -r dossier              # efface le dossier et tous ses sous-dossiers
```

Afficher et chercher

```
cat fic              # affiche le contenu d'un fichier
cat fic1 fic2 > fic3 # concatène fic1 et fic2 en fic3
grep "texte" fic     # affiche les lignes de fic contenant texte
wc fic               # compte les lignes/mots/caractères dans fic
```

RÉCURSIVITÉ, PILES, FILES, ARBRES

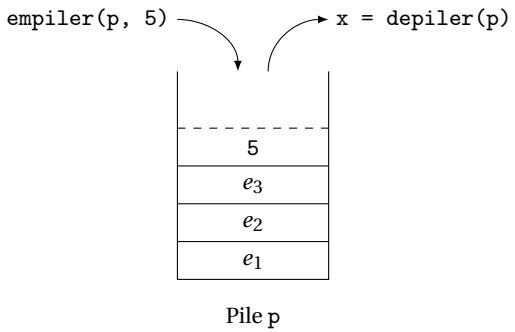
⬙ *Fonction récursive ou itérative*

Une fonction est dite **récursive** lorsqu'elle s'appelle elle-même. Par opposition, une fonction est dite **itérative** lorsqu'elle remplace un processus récursif par une boucle `while` ou `for`.

```
# Version récursive
def factorielle(n):
    if n == 0:
        return 1
    else:
        return n*factorielle(n-1)
```

```
# Version itérative
def factorielle(n):
    p = 1
    while n > 0:
        p = p * n
        n = n - 1
    return p
```

⬙ *Les piles* ⟶ **LIFO** (*Last In First Out : dernier entré, premier sorti*)

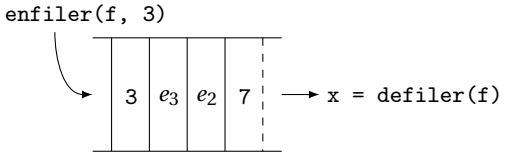


Pile p

```
# IMPLEMENTATION PAR FONCTION
def empiler(p, x):
    p.append(x)

def depiler(p):
    if len(p) > 0:
        return p.pop()
    else:
        return None
## Utilisation
p = [] # la pile est une liste
empiler(p, 5)
x = depiler(p) # x vaut 5
```

```
# IMPLEMENTATION PAR CLASSE
class Pile:
    def __init__(self):
        self.p = [] # p devient un attribut de la classe Pile

    def empiler(self, x):
        self.p.append(x)

    def depiler(self):
        if len(self.p) > 0:
            return self.p.pop() # retire et renvoit le dernier élément de la liste
        else:
            return None
## Utilisation
mapile = Pile()      # création d'un objet Pile via le constructeur de la classe
mapile.empiler(5)    # on empile la valeur 5 via la méthode de classe
x = mapile.depiler() # on dépile ce qu'il y a au sommet de la pile
```

⬙ *Les files* ⟶ **FIFO** (*First In First Out : premier entré, premier sorti*)



File f

On peut, suivant le même principe que pour une pile, implémenter une classe `File` *permettant une programmation objet du concept de file.*

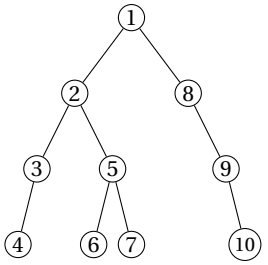
```
# IMPLEMENTATION PAR FONCTION
def enfiler(f, x):
    f.append(x)

def defiler(f):
    if len(f) > 0:
        return f.pop(0)
    else:
        return None
## Utilisation
f = [] # la file est une liste
enfiler(f, 3)
x = defiler(f) # x vaut 7
```

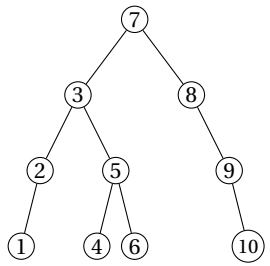
⬙ *Les arbres binaires*

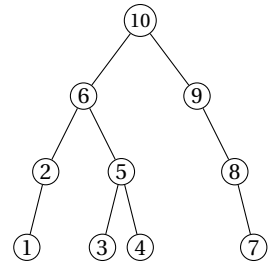
Un arbre binaire est un ensemble de **noeuds** possédant 3 attributs :

- une clef ou valeur ;
- un sous-arbre gauche ;
- un sous-arbre droit ;

```
class Noeud:
    def __init__(self, val, fg, fd):
        self.val = val
        self.fils_gauche = fg
        self.fils_droit = fd
```








```
def prefixe(a):
    if a != None:
        print(a.val)
        prefixe(a.gauche)
        prefixe(a.droit)
```

```
def infixe(a):
    if a != None:
        infixe(a.gauche)
        print(a.val)
        infixe(a.droit)
```

```
def postfixe(a):
    if a != None:
        postfixe(a.gauche)
        postfixe(a.droit)
        print(a.val)
```

La **taille** d'un arbre binaire est égale au nombre de noeuds de l'arbre.
La **profondeur** d'un noeud est le niveau du noeud en partant de la racine.
La **hauteur** d'un arbre binaire est égale à son nombre de niveaux.
Un **arbre binaire** est dit "**de recherche**" si pour chaque noeud, la valeur qui lui est associée est supérieure aux valeurs de son sous-arbre gauche et inférieure aux valeurs de son sous-arbre droit. Ainsi, parmi les trois arbre précédents, celui du milieu est un arbre binaire de recherche.

BASE DE DONNÉES ET REQUÊTES SQL

Schéma relationnel

Une *base de données relationnelle* est un système permettant de stocker et d'organiser des données dans des tableaux à deux dimensions appelés **tables** ou **relations**.
Les *lignes* de ces relations sont des **n-uplets** ou **enregistrements**.
Les *colonnes* sont appelés des **attributs** ou **champs**.

Une **clef primaire** est un attribut obligatoire qui permet d'identifier de manière unique un enregistrement dans une table.

Une **clef étrangère** est un attribut optionnel qui fait référence à une clef primaire dans une autre table : **son rôle est de relier deux tables entre elles**.

Langage SQL

```
-- Sélection
SELECT * FROM nom_table -- affiche tous les enregistrements d'une table

SELECT DISTINCT col FROM nom_table -- supprime les doublons
ORDER BY col ASC -- tri les résultats par ordre croissant, DESC pour décroissant

SELECT col1, col2 FROM nom_table -- n'affiche que certaines colonnes
WHERE condition -- opérateurs : =, !=, <=, LIKE, BETWEEN, IN, AND, OR, NOT

-- Jointure : mise en relation de 2 tables via clef primaire et étrangère
SELECT table1.col1, table2.col2 FROM table1 JOIN table2
ON table1.idp = table2.ide -- avec idp : clef primaire et ide : clef étrangère
WHERE condition            -- on peut rajouter des conditions si nécessaire

-- Insertion, modification et suppression
INSERT INTO nom_table VALUES (val1, val2, ...)
UPDATE nom_table SET col1 = ... WHERE id = ...
DELETE FROM nom_table WHERE id = ...

-- Mots-clefs
SELECT COUNT(*) -- compte les enregistrements sélectionnés
FROM nom_table
WHERE condition

SUM(col) -- additionne toutes les valeurs d'un champ numérique
AVG(col) -- calcule la moyenne des valeurs d'un champ numérique
MIN(col) -- renvoit la valeur minimale d'un champ numérique
MAX(col) -- renvoit la valeur maximale d'un champ numérique
NULL     -- signifie qu'il y a une absence de valeur
```