

# **Introduction au langage Python**

**LGLSI1- ISTIC**

**Janvier 2020**

# **Introduction au langage Python**

**Cours 1/4**

**Les bases**

**Janvier 2020**

# Algorithmie vs. Programmation

## Algorithmie

- Solution « informatique » relative à un problème
- Suite d'actions (instructions) appliquées sur des données
- 3 étapes principales :
  1. saisie (réception) des données
  2. Traitements
  3. restitution (application) des résultats

## Programme

- Transcription d'un algorithme avec une syntaxe prédéfinie
- **Python**
- Même principes fondamentaux que les autres langages objets (Delphi, Java, C#, etc.)
- Python s'enrichit de bibliothèques de calcul spécialisées (mathématique, bio informatique, etc.)

## Mode compilé vs. mode interprété

Langage interprété : + portabilité application ; - lenteur (R, VBA, **Python**...)

Langage compilé : + rapidité ; - pas portable

(solution possible : write once, compile anywhere ; ex. Lazarus)

Langage pseudo-compilé : + portabilité plate-forme ; - lenteur (?)

(principe : write once, run anywhere ; ex. Java et le principe JIT)



**Python** est interprété, il est irrémédiablement lent, mais...

on peut lui associer des librairies intégrant des fonctions compilées qui, elles, sont très rapides.

Cf. « [Why Python is Slow](#) », « [PythonSpeed – Performance Tips](#) ».

## Etapes de la conception d'un programme (Génie Logiciel)

1. **Déterminer les besoins et fixer les objectifs** : que doit faire le logiciel, dans quel cadre va-t-il servir, quels seront les utilisateurs types ? On rédige un cahier des charges avec le commanditaire du logiciel (Remarque : commanditaire = **maître d'ouvrage** ; réalisateur = **maître d'œuvre**)
2. **Conception et spécifications** : quels sont les fonctionnalités du logiciel, avec quelle interface ?
3. **Programmation** : modélisation et codage
4. **Tests** : obtient-on les résultats attendus, les calculs sont corrects, y a-t-il plantage et dans quelles circonstances ? (tests unitaires, tests d'intégration, etc.)
5. **Déploiement** : installer le chez le client (vérification des configurations, installation de l'exécutable et des fichiers annexes, etc.)
6. **Maintenance** : corrective, traquer les bugs et les corriger (patches) ; évolutive (ajouter des fonctionnalités nouvelles au logiciel : soit sur l'ergonomie, soit en ajoutant de nouvelles procédures)

## – Introduction sur un exemple –

Un exemple de programme en Python 3 :

```
def puissance(x,n):
    res = 1
    i = 1
    while i <= n:
        res = res*x
        i = i + 1
    return res

y = puissance(12,6)
print(y)
```

C'est un programme un peu avancé ...

## – Introduction sur un exemple –

Un exemple de programme en Python 3 :

```
def puissance(x,n):
    res = 1
    i = 1
    while i <= n:
        res = res*x
        i = i + 1
    return res

y = puissance(12,6)
print(y)
```

C'est un programme un peu avancé... mais essayons de comprendre ce qu'il fait.

## – Les différents mots –

```
def puissance(x,n):
    res = 1
    i = 1
    while i <= n:
        res = res*x
        i += 1
    return res

y = puissance(12,6)
print(y)
```

Les couleurs sont un ajout pratique pour la lisibilité, elle peuvent changer d'un éditeur à l'autre. Ici on a :

- en orange des mots-clés du langage Python
- en violet des fonctions du langage Python
- en bleu des noms choisis par le programmeur

## – Les différents mots –

```
def puissance(x,n):
    res = 1
    i = 1
    while i <= n:
        res = res*x
        i = i + 1
    return res

y = puissance(12,6)
print(y)
```

Les mots de Python, en orange et en violet, sont en anglais.

<b>def</b> = define = définir	<b>while</b> = tant que
<b>return</b> = retourner	<b>print</b> = imprimer

## – L'indentation –

```
def puissance(x,n):
    res = 1
    i = 1
    while i <= n:
        res = res*x
        i = i + 1
    return res

y = puissance(12,6)
print(y)
```

- Les lignes ne commencent pas toutes au même endroit.
- Ce n'est pas un détail!
- Le positionnement (on dit l'indentation) des lignes est important pour leur signification.

Si on masque temporairement les lignes indentées, on obtient :

```
def puissance(x,n):
    ...
y = puissance(12,6)
print(y)
```

```
def puissance(x, n):  
    ...  
y = puissance(12, 6)  
print(y)
```

- **def** définit une **fonction** qui s'appelle ici **puissance**
- La fonction **puissance(x,n)** calcule la valeur  $x^n$  (on verra comment plus tard)
- **puissance(12,6)** appelle la fonction **puissance** avec les paramètres 12 et 6, qui calcule donc  $12^6$
- **y = puissance(12,6)** stocke la valeur  $12^6$  dans **y**
- **print** est une **fonction** de Python qui affiche la valeur de son paramètre à l'écran

*Essayons un peu le programme et des variantes ...*

## – La fonction puissance –

```
def puissance(x, n):
    res = 1
    i = 1
    while i <= n:
        res = res * x
        i = i + 1
    return res
```

L'indentation signifie que toutes les lignes font partie de la fonction puissance

- La fonction prend deux paramètres  $x$  et  $n$
- Au début on met la valeur 1 dans `res` et `i`
- | **while** (= tant que) possède
  - une condition  $i \leq n$
  - deux lignes indentées
  - ces deux lignes sont répétées “tant que” la condition est vraie
- **return** (= retourner) termine la fonction en renvoyant la valeur de `res`

## – Détail de la boucle while –

```
while i <= n:  
    res = res * x  
    i = i + 1
```

Pour l'exemple **x** = 2 et **n** = 4  
Au début **res** et **i** valent 1

	<b>x</b>	<b>n</b>	<b>res</b>	<b>i</b>
<b>Avant de commencer</b>	2	4	1	1
<b>Etape 1</b>	2	4	2	2
<b>Etape 2</b>	2	4	4	3
<b>Etape 3</b>	2	4	8	4
<b>Etape 4</b>	2	4	16	5

A la fin, “**return res**” retourne donc la valeur 16 qui est bien  $2^4$

## – Robustesse du programme –

Qu'est-ce qui se passe si `n = 0` ? ou `x = 4.5` ? ou `n = -4` ?

On va corriger le problème si `n` est négatif, en rajoutant un `test` et en traitant le cas séparément. On va ajouter au début de `puissance` :

```
if n < 0:  
    print('erreur : pas de n négatif')  
    return
```

- `if` (= si) est un test, ici on regarde si `n` est strictement négatif
- les deux lignes indentées ne sont effectuées que si la condition du `if` est vraie
- si `n < 0`, on affiche un message d'erreur et `return` termine la fonction en ne retournant rien

## – La nouvelle fonction puissance –

```
def puissance(x,n):      # calcule x puissance n
    if n < 0:            # cas n < 0 non géré
        print('erreur : pas de n négatif')
        return
    res = 1
    i = 1
    while i <= n:        # on fait n fois
        res = res*x       # multiplier res par x
        i = i + 1          # ajouter 1 à i
    return res
```

- On a ajouté des **commentaires**:
  - le caractère **#** indique que la suite de la ligne est un **commentaire**
  - les **commentaires** sont ignorés par Python
  - ils servent à décrire le programme pour les êtres humains qui le lisent
- Les commentaires sont **très importants** en programmation... on y reviendra

## – En résumé –

- On a vu un exemple avancé, qui nous a permis de voir, dans les grandes lignes, à quoi ressemble un programme
- On a vu
  - que les lignes sont des instructions à effectuer, dans l'ordre
  - qu'on peut stocker des valeurs (dans `y`, `x`, ...)
  - qu'on peut effectuer des calculs (`res * x`)
  - que des lignes peuvent être effectuées plusieurs fois (`while`), ou seulement si une condition est vérifiée (`if`)
  - qu'on peut créer des fonctions (`puissance`) ou utiliser des fonctions de Python (`print`)
  - ...
- Dans la suite, on va apprendre pas à pas et dans le détail toutes ces notions (et bien d'autres), afin que vous maîtrisiez les bases de la programmation

# Variables, types et opérations

## – Types de valeurs –

- Les valeurs de base possèdent un **type**
- Le **type** va notamment déterminer ce qui se passe quand on fait une **opération** sur des valeurs

Les principaux **types** :

- **entier (int)** : 12      -4      123545      ...
- **flottant (float)** : 3.14159      -1.5      12.      4.56e12      ...
- **booléen (bool)** : **True** (vrai) ou **False** (faux)
- **indéfini, rien** : **None**
- **chaîne de caractères (str pour "string")** : '**chaîne de caractères**'  
**'IUT info'**, ...



Les majuscules/minuscules sont importantes :  
**True** 6 = **true**

## – Transtypage –

- La fonction **type()** permet de connaître le type d'une valeur
- On peut demander à Python de changer le type d'une valeur
- On peut par exemple toujours transformer une valeur de base en chaîne de caractères avec la fonction **str()**
- Par exemple **str(51)** renvoie la chaîne '**51**'
- **Attention :** le nombre 51 et la chaîne '**51**' ce n'est **pas la même chose** pour Python. On y reviendra.
- **int()** convertit en entier, quand cela est possible
- **float()** convertit en flottant, quand cela est possible
- **bool()** convertit en booléen

## – Quelques exemples –

`int(4.5) ! 4`

`int('IUT') ! erreur`

`str(4) ! '4'`

`bool(4) ! True`

`int(-4.5) ! -4`

`float(4) ! 4.`

`str(True) ! 'True'`

`bool(0) ! False`

`int('0345') ! 345`

`float('4.5') ! 4.5`

`str(-4.5) ! '-4.5'`

`bool('IUT') ! True`

En pratique, on se sert surtout de :

- `str` qui fonctionne tout le temps
- `int` et `float` appliqués à une chaîne de caractères qui correspond à un nombre
- `int` appliqué à un `float` pour tronquer les décimales

## – Opérations sur les nombres –

- Sur les **int** et sur les **float** on a l'addition **+**, la soustraction **-**, la multiplication **\*** et la division **/**
- Si on compose deux **int** on obtient un **int**, sauf la division qui renvoie un **float**
- Si on compose deux **float**, ou un **int** et un **float**, on obtient un **float**
- On dispose également de la **division Euclidienne**, avec quotient et reste comme en primaire. Le quotient de **x** et **y** est **x // y** et leur reste est **x % y**
- Il y a enfin l'opération **puissance** qui se note **x \*\* y**
- Les opérations suivent les règles de priorités usuelles et on peut utiliser des parenthèses: **(4+2)\*1.2**

## – Opérations avec booléens –

- On a les opérations sur les booléens :
  - **and** c'est le **ET** logique, **x and y** vaut **True** seulement quand **x** et **y** valent **True**
  - **or** c'est le **OU** logique, **x or y** vaut **False** seulement quand **x** et **y** valent **False**
  - **not** c'est la **négation** logique, **not (True) = False** et **not (False) = True**
- Les comparaisons produisent des booléens :
  - Le test d'**égalité** se fait avec **==**
  - Le test de **différence** se fait avec **!=**
  - On a aussi **< <= > >=** pour comparer selon l'ordre usuel (**ordre du dictionnaire** pour les chaînes)

## – Opérations sur les chaînes de caractères –

- Si on utilise **+** sur deux chaînes de caractères, on effectue la concaténation des deux chaînes :  
**'IUT' + 'info'** !   **'IUTinfo'**
- Si on “multiplie” une chaîne par un entier **n**, on la répèren fois :  
**'IUT' \* 3** !   **'IUTIUTIUT'**

## – Autres opérations –

- Il existe beaucoup d’autres opérations sur les chaînes
- On a accès à plein d’opérations mathématiques (cosinus, . . . )
- On verra ça plus tard dans le semestre

## – Ecriture condensée –

- On a souvent besoin d'**ajouter** une valeur dans une variable, ce que l'on a fait avec **x = x + y**
- Il existe en Python (et dans beaucoup d'autre langages) une écriture **plus compacte** pour faire la même chose : **x += y**
- On peut l'utiliser avec d'autres opérations, et sur différents type.
  - Pour **x entier** et **s chaîne de caractères**, on a :

<b>x += 3</b>	→	ajoute 3 à x
<b>x *= 2</b>	→	multiplie x par 2
<b>x //= 4</b>	→	x est changé en son quotient par 4
<b>s += 'toto'</b>	→	concatène ' <b>toto</b> ' à la fin de s
<b>s *= 3</b>	→	remplace s par 3 copies de s
<b>s // 4</b>	→	erreur

- *il n'y a pas de notation i++ en Python*

## – Nommage –

Dans le programme d'introduction, on a utilisé nos propres noms, en bleu :

```
def puissance(x,n):  
    ...  
y = puissance(12,6)  
print(y)
```

Les **règles** de nommage pour ce cours sont les suivantes :

- le caractère “**underscore**” (le tiret bas de la touche 8) est considéré comme une lettre
- on n'utilise **jamais** d'accent, de cédille, . . .
- Les noms commencent par une **lettre** majuscule ou minuscule, puis sont composés de **lettres et de nombres**:  
exemple      `_ex2`      `Ex2mpl1`      `2013iut`
- les **mots réservés** de Python sont interdits
- il y a aussi des **conventions**, *plus tard* ...

## – Mots réservés –

Les mots suivants sont réservés pour le langage :

and	as	assert	break	class	continue
def	del	elif	else	except	finally
for	from	global	if	import	in
is	lambda	nonlocal	not	or	pass
raise	return	try	while	with	yield

- On n'utilisera pas non plus comme nom : **True** , **False** , **None**
- Pour voir la liste des mots réservés, dans un terminal Python taper :  
**import keyword**  
**print(keyword.kwlist)**

## – Variables –

- une **variable** est un **nom** qui référence une valeur dans la mémoire
- on peut s'en servir dans les **calculs**
- elle a le **même type** que la valeur qu'elle référence

## – Affectation –

- L'**affectation** d'une variable consiste à lier un **nom** à une **valeur**
- La syntaxe : **nom = expression**, où **expression** est une **valeur** ou un **calcul** qui produit une valeur :  
 $x = 3$        $y = 'IUT'$        $z = x + 2$
- On peut **affecter à nouveau** une même variable, on perd le lien avec l'ancienne valeur



Ce n'est pas **du tout** le = des mathématiques. Il faut le lire comme "prend la valeur" :  $x = x + 1$

## – Etapes de l'affectation –

$$x = 40 + 2$$

- On commence par calculer le **membre droit**, ici on trouve **42**

42

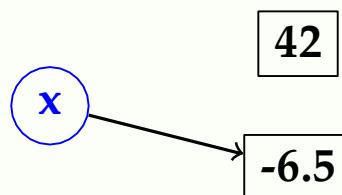
- Ensuite on crée le nom pour **x** (sauf s'il a déjà été créé)



- Enfin on relie la **variable** à sa **valeur**



- En cas de **réaffectation**, le lien d'avant est **perdu**: **x= -6.5**



# Instructions et blocs

## – Instructions et séquence d’instructions –

```
print('a x^2 + b x + c = 0')
a = float(input('a = '))
b = float(input('b = '))
c = float(input('c = '))
delta = b*b-4*a*c
if delta > 0:
    print('deux solutions')
elif delta == 0:
    print('une solution')
else:
    print('pas de solution')
```

- Comme on a vu dans l’introduction, les **instructions** sont effectuées dans l’ordre, de **haut en bas**
- En Python, il n’y a **qu’une instruction par ligne**
- Le flot d’instructions peut-être modifié / redirigé par des conditions (**if**), des boucles (**while**), . . .

### – Au passage ... input –

```
a = float(input('a = '))
```

- On a utilisé une nouvelle fonction, la fonction **input(str)**
- Cette fonction permet à l'utilisateur de **saisir une valeur au clavier**
- Quand on écrit **a= input('valeur= ')**, la chaîne '**valeur =**' est affichée à l'écran (comme avec **print**) et le programme attend que soit rentré une valeur, qu'il met dans la variable **x**, c'est une affectation normale
- La fonction **input** renvoie **toujours** une chaîne de caractères
- On a donc utilisé le **transtypage** avec la fonction **float**

## – Blocs d'instructions –

Certaines instructions sont regroupées en blocs de la façon suivante :

**entête du bloc:**

**instruction 1 du bloc**

**instruction 2 du bloc**

**instruction 3 du bloc**

**instruction hors bloc**

- L'**indentation** (le décalage) se fait avec la **tabulation** (la touche au-dessus du capslock sur le clavier, cf TP)
- On peut **insérer** un bloc dans un bloc, un bloc dans un bloc dans un bloc, . . .
- L'**indentation** fait partie du langage Python, changer l'indentation **change la signification** du programme

# Instruction conditionnelle (if)

## – La conditionnelle : le if –

```
if delta > 0:  
    print('deux solutions')  
elif delta == 0:  
    print('une solution')  
else:  
    print('pas de solution')
```

- Sur l'exemple on commence par tester si **delta** > 0
  - Si c'est le cas, on effectue le bloc qui suit, et on affiche **deux solutions**
- Sinon, on teste si **delta** == 0
  - Si oui, on indique qu'il y a une seule solution
- Sinon on indique qu'il n'y a pas de solution

## – La conditionnelle : le if –

La forme la plus simple est

```
if expression:  
    instruction 1 du if  
    instruction 2 du if  
    ...  
    instruction après if
```

- **expression** est une expression qui retourne un booléen, qui est donc évaluée à **True** ou **False**
- les instructions du **bloc du if** sont effectuées **uniquement si** l'expression est évaluée à **True**
- dans tous les cas, le programme reprend à l'**instruction après if**

## – La conditionnelle : le **if** avec **else** –

La forme avec **else** (= sinon) :

```
if expression:  
    instruction 1 du if  
    ...  
else:  
    instruction 1 du else  
    ...  
instruction après if/else
```

- les instructions du **bloc du if** sont effectuées **uniquement si** l'expression est évaluée à **True**
- les instructions du **bloc du else** sont effectuées **uniquement si** l'expression est évaluée à **False**
- dans tous les cas, le programme continue à l'**instruction après if/else**

## – La conditionnelle : le **elif** –

La forme avec **elif** (= contraction de else et if) :

```
if expression1:  
    bloc du if  
elif expression2:  
    bloc du elif  
else:  
    bloc du else  
instruction après if/elif/else
```

- les instructions du **bloc du if** sont effectuées **uniquement si expression1 vaut True**
- les instructions du **bloc du elif** sont effectuées **uniquement si expression1 vaut False et expression2 vaut True**
- les instructions du **bloc du else** sont effectuées **uniquement si expression1 vaut False et expression2 vaut False**

## – La conditionnelle : le **elif** –

La forme avec **elif** (= contraction de else et if) :

```
if expression1:  
    bloc du if  
elif expression2:  
    bloc du elif  
else:  
    bloc du else  
instruction après if/elif/else
```

- les instructions du **bloc du if** sont effectuées **uniquement si expression1** vaut **True**
- les instructions du **bloc du elif** sont effectuées **uniquement si expression1** vaut **False** et **expression2** vaut **True**
- les instructions du **bloc du else** sont effectuées **uniquement si expression1** vaut **False** et **expression2** vaut **False**
- On peut mettre **plusieurs elif**, les conditions sont évaluées **dans l'ordre**, et seule **la première** qui vaut **True** est considérée

# Les slices

## – Notion de slice –

- On a vu qu'on peut accéder au  $i$ -ème élément d'une liste ou d'une chaîne avec `t[i]`
- Le **slice** consiste à accéder à une **portion** d'une liste ou d'une chaîne
- **Notation :** `t[debut:fin]` prend la sous-liste où la sous-chaîne comprise entre les indices **debut** et **fin-1**
- **Attention :** c'est **fin -1** comme pour les **range**

```
s = 'bonjour'  
print(s[2:5]) !      'njo'
```

## – Indices négatifs –

- On peut utiliser des **indices négatifs**
- L'indice **-i** est le même que **len - i**

0	1	2	3	4	5	6
<b>b</b>	<b>o</b>	<b>n</b>	<b>j</b>	<b>o</b>	<b>u</b>	<b>r</b>
-7	-6	-5	-4	-3	-2	-1

```
s = 'bonjour'  
print(s[-2]) ! u  
print(s[1:-2]) ! onjo
```

## – Paramètres par défaut –

- Dans un **slice**, le début est par défaut 0 : **t[:4]** est la même chose que **t[0:4]**
- Dans un **slice**, la fin est par défaut **len(t)** : **t[4:]** est la même chose que **t[4:len(t)]**
- **t[:7]** ce sont donc les 7 premiers éléments
- **t[2:]** ce sont les éléments à partir du troisième (le premier est à 0)
- **t[:-2]** ce sont tous les éléments sauf les 2 derniers
- **t[-5:]** ce sont les 5 derniers éléments



Lors d'un **slice**, Python recopie la portion de chaîne (ou de liste, ...) qui est extraite.

# **Notion d'exception**

## – Qu'est-ce qu'une exception ? –

- Quand un programme plante, c'est qu'il y a eu un problème qui a levé une exception
- Le mécanisme d'exception sert à signaler une anomalie de fonctionnement
- Quand une telle anomalie se produit, on peut dans le programme :
  - ne rien faire et laisser le programme planter
  - interceppter l'exception et traiter le problème dans le programme

Message d'erreur type :

Traceback (most recent call last):

File "code4/erreur.py", line 1, in <module>

x = 3 // 0

**ZeroDivisionError:** integer division or modulo by zero

## – Mécanisme d'interception –

- Pour **intercepter une exception**, il faut mettre le code qui risque d'en générer une dans un bloc **try**:
- L'interception se fait ensuite dans un bloc **except**:

**try:**  
    instructions à risque

**except:**  
    instructions en cas d'exception

### – Exemple : saisir un entier –

- la fonction demande un nombre et tente de le convertir en entier avec la fonction `int()`
- si elle n'y arrive pas, une exception est levée, qui est interceptée avec le `except`
- en cas de problème on retourne `None` : le programme ne plante pas, on peut redemander le nombre

```
def saisieNombre ( s='nombre = ' ):  
    try :  
        return int ( input ( s ))  
    except :  
        return None
```

## – Interception (suite) –

- Un même code peut générer **plusieurs types d'erreurs**
- Il peut être utile de savoir les **distinguer** lors de l'interception.

```
try :  
    n = int(input('nombre = '))  
    print(1 / n)  
except :  
    print('il y a eu une erreur')
```

## – Noms d'exceptions –

- On peut paramétriser les **except** avec un nom d'exception
- Si un **except** est paramétré, il n'est exécuté que si une exception **du bon nom** est levée
- Le **nom** de l'exception est celui indiqué sur la dernière ligne du message d'erreur

Traceback (most recent call last):

  File "code4/erreur.py", line 1, in <module>

    x = 3 // 0

**ZeroDivisionError:** integer division or modulo by zero

## – Exemple avec plusieurs except –

```
try :  
    n = int(input('nombre = '))  
    print(1 / n)  
except ZeroDivisionError:  
    print('Division par zero !')  
except:  
    print('il y a eu une erreur')
```

## – Lever sa propre exception –

- Il est possible de lever volontairement une exception pour signaler un problème
- L'instruction est `raise NameError(str)`, où `str` est un message d'information
- **Attention :** le nom d'une telle exception est `NameError`

```
def puissance(x, n):  
    if n < 0:  
        raise NameError('pas de puissance négative')  
    r = 1  
    for i in range(n):  
        r *= x  
    return r
```

# Introduction au langage Python

Cours 2/4

Boucles, listes et  
fonctions

Janvier 2020

# La boucle `while`

## – Exemple –

```
res = 1
i = 1
while i < n:
    res = res ← x
    i = i + 1
print(res)
```

- on avait vu le **while** lors du cours 1
- c'est une **instruction de boucle** : son bloc associé est répété tant que la **condition est vraie**

**while condition:**

**instruction 1 du while**

**instruction 2 du while**

...

**instruction après while**

- chaque exécution de la séquence d'instructions du **while** est appelé une **itération**

## – Utilisation : faire un nombre fixé de fois –

**i = 1**

**while i <= n:**

**instruction 1 du while**

**instruction 2 du while**

...

**i = i + 1**

**instruction après while**

	<b>i</b>
<b>Avant de commencer</b>	<b>1</b>
<b>Fin itération 1</b>	<b>2</b>
<b>Fin itération 2</b>	<b>3</b>
...	
<b>Fin itération n-1</b>	<b>n</b>
<b>Fin itération n</b>	<b>n+1</b>

## – Deux exemples –

```
nbr = int(input('Combien ? '))
i = 1
while i <= nbr:
    print('bonjour')
    i = i + 1
print('Fini !')
```

```
from time import sleep
nbr = int(input('Combien de secondes ? '))
i = nbr
while i > 0:
    print(str(i) + ' . . . ')
    sleep(1)
    i = i - 1
print('BOOOM !')
```

## – Nombre d’itérations non fixé –

Redemander un nombre tant que nécessaire :

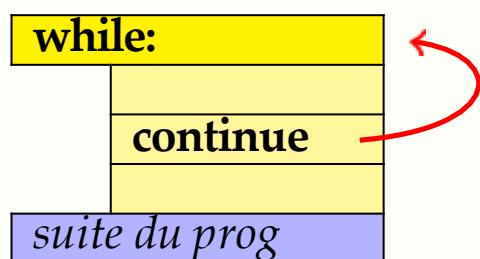
```
a = int(input('Nombre entre 1 et 10 : '))
while a < 1 or a > 10:
    a = int(input('Erreur, entre 1 et 10 : '))
print('Bravo, votre nombre est ' + str(a))
```

Combien de tirages de deux dés pour faire 12 :

```
from random import randint
des = 0
nbr = 0
while des != 12:
    des = randint(1,6) + randint(1,6)
    nbr = nbr + 1
print(str(nbr) + ' tirages pour faire 12')
```

## – break et continue –

- Il existe deux moyens de modifier le **flux normal** des itérations d'une boucle
- Ces moyens sont **à éviter**, sauf quand ils rendent le programme plus clair (ce qui arrive)
- l'instruction **break** arrête la boucle, et continue donc l'exécution du programme après la boucle
- l'instruction **continue** arrête l'itération en cours, et reprend à l'itération suivante de la boucle



## – Un exemple avec **break** –

```
from random import randint
de = randint(1,100)
while True:
    a = int(input('Devinez un nombre (1-100) : '))
    if a == de:
        break
    if a < de:
        print('trop petit')
    if a > de:
        print('trop grand')
print('Bravo !')
```

- **while True** boucle indéfiniment, la condition est toujours vérifiée
- On utilise **break** pour arrêter les itérations quand le nombre est deviné

## – Boucles infinies –



Si la condition de boucle est toujours vraie, et qu'il n'y a pas de **break** pour en sortir, le programme **reste bloqué en tournant indéfiniment dans la boucle !** C'est une erreur classique.

- Pour forcer l'arrêt d'un programme, dans idle ou un terminal python faire control+C
- Pour forcer la fermeture faire control+D

## – Les boucles imbriquées –

- On peut utiliser des **boucles dans des boucles**, des boucles dans des boucles ...
- Cela arrive fréquemment quand on veut gérer des objets à **deux dimensions**, mais aussi dans d'autres situations (voir les algorithmes de tri dans l'autrecours)

```
i = 1
while i < 5:
    j = 1
    while j < 4:
        print(' (' + str(i) + ', ' + str(j) + ') ')
        j = j + 1
    i = i + 1
```

# **Les listes**

## – Présentation –

- La **liste** est un **type avancé de données**, elle sert à stocker une séquence de valeurs
- On peut créer une liste par une **affectation normale**, où on met entre crochets et séparés par des virgules les différentes valeurs de la liste

```
lst = [ 3, 'toto', 4.5, False ]
```

- Il y a une liste particulière, la **liste vide []** qui ne contient aucun élément
- On peut accéder au *i*-ème élément d'une liste en utilisant les crochets, le *i*-ème élément de lst est lst[i]
- **Attention :** les indices commencent à 0 et non à 1 !

```
lst = [ 3, 'toto', 4.5, False ]
print( lst[1] )
>>> 'toto'
```

## – Affectations et listes –

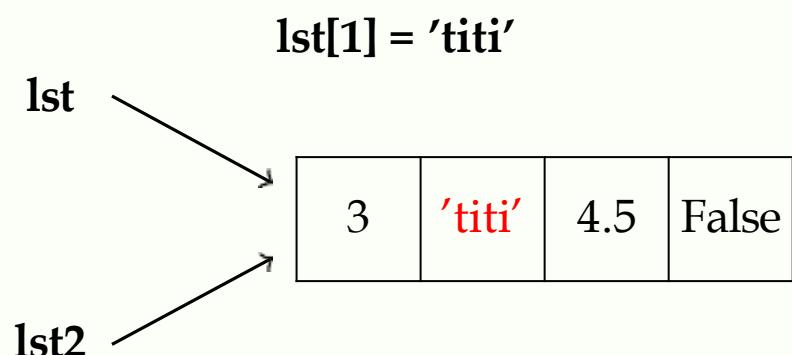
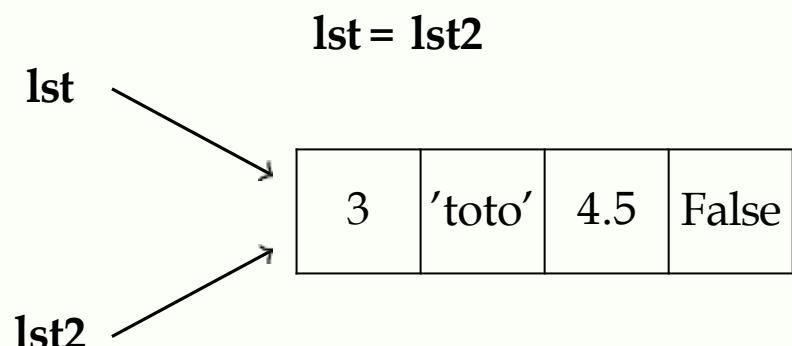


Il y a des subtilités ici, à bien travailler.

- Une liste est un objet **modifiable**, on peut modifier ses valeurs, ses éléments, etc sans créer de nouvelle liste
- Ce n'est pas vrai pour les autres types qu'on a vu, notamment les chaînes de caractères
- On peut changer la *i*-ème valeur de `lst` avec une affectation classique : `lst[2] = 'titi'`
- Si `lst` est une liste, `lst2 = lst` associe au nom `lst2` la même liste (qui est modifiable).
- En conséquence, si on modifie ensuite `lst`, **on modifie aussi `lst2`!**

```
lst = [ 3, 'toto', 4.5, False ]  
lst2[1] = 'titi'  
print(lst)  
>>> [ 3, 'titi', 4.5, False ]
```

## – Affectations et listes –



## – Quelques opérations –

- `len( lst )` retourne la longueur de `lst`
- `x in lst` renvoie un booléen qui est `True` quand `x` est dans `lst`
- Le `+` crée une **nouvelle liste** qui est la concaténation de deux listes
  - $[ 3, 4 , 7] + [ 'toto' , 5]$   
`>>> [ 3, 4, 7, 'toto', 5]`
- Si on multiplie une liste par un entier  $n$ , cela crée une **nouvelle liste** où l'ancienne est répétée  $n$  fois :

```
lst = [ 3, 4 , 7]
lst2 = lst * 3
print(lst2)
>>> [ 3, 4, 7, 3, 4, 7, 3, 4, 7]
```

## – Autres opérations –

- Les liste sont des **objets**, une notion hors-programme. On peut néanmoins utiliser certaines fonctions rattachées aux listes (on appelle cela des méthodes)
- Pour utiliser une telle fonction sur une liste **lst**, on utilise **lst.nom()**
- **Attention :** en générale cela **modifie** la liste

Quelques exemples :

- **lst.append(x)** ajoute la valeur de **x** à la fin de **lst**
- **lst.extend(lst2)** ajoute tous les éléments de **lst2** à la fin de **lst**
- **lst.pop()** supprime le dernier élément de **lst** et retourne sa valeur
- **lst.pop(i)** supprime le *i*-ème élément de **lst** et retourne sa valeur
- ...

## – Parcourir une liste –

- Une façon naturelle de **parcourir** une liste est d'utiliser **une boucle while**, en faisant varier l'indice dans la liste

```
lst = [1, 'toto', 4.5, False]
i = 0
while i < len(lst):
    print(lst[i])
    i = i + 1
```

- Il existe d'autre moyens de parcourir une liste, que l'on verra au prochain cours

# Les chaînes de caractères

## – Déclaration de chaînes de caractères –

- On peut déclarer une chaîne entre **apostrophes** comme on a fait jusqu'ici : `x = 'toto'` ...
  - ou entre **guillemets** : `x = "toto"`
- les deux sont valides, on peut par exemple utiliser la première quand il y a des guillemets dans la chaîne et la seconde quand il y a des apostrophes.
- Comment faire s'il y a à la fois des ' et des " ? on utilise les caractères spéciaux \\'et " :  
`s = ' il a dit : "à l'abordage !'"`
- **Attention :** \\' est un seul caractère" (ce sont des caractères spéciaux) :

`len('d\'abord')` →

7

## – Déclaration sur plusieurs lignes –

- On peut déclarer une chaîne **sur plusieurs lignes** en utilisant des triples apostrophes ou triples guillemets comme délimiteurs :  
`s= """Ceci est une  
chaine sur  
plusieurs lignes."""`
- Les saut de lignes seront **encodés** par le caractère `\n`
- On peut également utiliser **juste un backslash** `\` avant la fin de ligne et continuer sur la ligne suivante `s= 'Ceci est une\  
chaine sur \  
plusieurs lignes.'`

## Une chaîne de caractères est une liste particulière avec des méthodes associées

```
#définir une chaîne
```

```
s1 = "bonjour le monde" print(s1)
```

Guillemets pour délimiter une chaîne

```
#longueur  
long = len(s1)  
print(long)  
#accès indicé  
s2 = s1[:7]  
print(s2)  
#non modifiable  
print(s)  
#recherche d'une sous-chaîne id = s.find("O")
```

Mécanisme identique aux tuples et listes

ERREUR. Une chaîne n'est pas modifiable. Il faut mettre le résultat d'une manipulation dans une autre chaîne.

```
print(id) → 3 (1ère occurrence si plusieurs)  
#nb d'occurrences nb = s.count("ON")  
print(nb) → 2  
#remplacement de « O » par « A »  
SA = s.replace("O","A") print(SA)
```

Des méthodes spécifiques permettent de manipuler les chaînes. Cf. <https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str>

## Transformation explicite en liste (pour traitements)

Une chaîne peut être transformée en liste pour réaliser des traitements sophistiqués. L'outil est très souple.

```
#transf. en liste  
liste = list(S)  
print(liste) ←
```

[ 'B', 'O', 'N', 'J', 'O', 'U', 'R', ' ', 'L', 'E', ' ', 'M', 'O', 'N', 'D', 'E' ]

Toutes les opérations sur les listes sont possibles par la suite.

```
#découpage par séparateur  
decoupe = S.split(" ")  
print(decoupe) ←
```

[ 'BONJOUR', 'LE', 'MONDE' ]

Espace est utilisé comme séparateur ici, mais ça peut être tout autre carac., y compris un caractère spécial (ex. \t pour tabulation)

```
#former une chaîne à  
#partir d'une liste  
SB = "-".join(decoupe)  
print(SB) ←
```

"BONJOUR-LE-MONDE"

Les mots de la liste ont été fusionnés avec le séparateur "-". Tout séparateur est possible, y compris la chaîne vide.

## Un exemple

```
chaines_exemple.py - D:/_Travaux... - □ X
File Edit Format Run Options Window Help
# -*- coding: utf -*-
#mot à traiter
mot = input("mot à traiter : ")

#passer en maj.
mot = mot.upper()

#compter le nombre
#d'apparition de "S"
nb = mot.count("S")

#test
if (nb == 0):
    print("S n'est pas présent")
else:
    print("S est apparu {} fois".format(nb))

#fin du prog.
```

```
Python 3.4.3 Shell - □ X
File Edit Shell Debug Options Window Help
>>>
mot à traiter : saucisse
S est apparu 3 fois
>>> | Ln: 178 Col: 4
```

## – Caractères spéciaux –

- Voilà quelques caractères spéciaux utiles :

\'	apostrophe	\"	guillemet
\n	saut de ligne	\t	tabulation
\\"	antislash		

- Par exemple la chaîne `x = ' toto\nba'` est une chaîne de longueur 6, si on fait `print(x)` ils'affiche ...

```
>>> print(x)
```

**tota**

**ba**

## – Les chaînes sont non-modifiables –

- **Important :** une chaîne n'est pas modifiable.
- Si `x` contient une valeur de type `str` et que vous voulez la changer, il faut faire une nouvelle affectation de `x` :  
`x = 'toto'`  
`x[0] = 'p'` → **erreur** on ne peut pas modifier une chaîne  
`x = 'poto'` → on crée une nouvelle chaîne '`poto`'
- **Rappel :** c'est le contraire avec les listes :  
`lst = [1,4,6,7]`  
`lst[0] = 3` → `lst` vaut `[3,4,6,7]`

## – Opérations sur les chaînes –

- On a déjà vu la concaténation + de deux chaînes et la “multiplication” par un entier
- On a accès au *i*-ème caractère de la chaîne **s** avec **s[i]** (les indices commencent à 0)
- **len(s)** retourne la longueur de **s**

Il y a beaucoup d'autres opérations sur les chaînes, avec la notation **s.fonction()** notamment :

- **s.lower()** renvoie une nouvelle chaîne où les majuscules ont été changées en minuscules
- **s.upper()** renvoie une nouvelle chaîne où les minuscules ont été changées en majuscules
- **s.split(t)**, où **t** est une chaîne, renvoie un tableau de chaînes obtenues en **coupant s aux occurrences de t**
- ...

# Structures itérables et boucles for

## – Structure itérable –

- Une **structure itérable** est une structure qui contient plusieurs valeurs avec
  - une **valeur initiale**
  - une notion de **valeurs suivante**
- On connaît déjà deux exemples de structures itérables : les **chaînes de caractères** et les **listes** :  
`s = "abcdef"`  
`lst = [1, 4, 56, 2]`
- On peut changer un itérable en la liste, dans l'ordre, de ses éléments avec l'instruction **list( )**

## – les range –

- Une autre structure itérable très utilisée est retournée par la fonction **range( )**
- **range(a,b)**, où **a** et **b** sont des entiers, est un itérable qui commence à **a** et qui s'arrête à **b-1** :
  - **list(range(1,5)) — ! [1,2,3,4]**
- **range(b)** est une version condensée de **range(0,b)**
- **range(a,b,c)** est l'itérable qui commence à **a** et avance de **c** en **c** jusqu'à arriver en **b** (exclu)  
**list(range(1,7,2)) → [1,3,5]**
- **attention** on s'arrête avant **b** dans tous les cas

## – les boucles for –

- Comme **while**, l'instruction **for** est une **instruction de boucle**
- Elle permet de **parcourir** un itérable, dans l'ordre, en commençant au premier élément et en allant de **suivant en suivant**
- La syntaxe est la suivante

```
for x in iterable:  
    instruction 1 du for  
    instruction 2 du for  
    ...  
    instruction n du for  
suite du programme
```

## – Faire une action $n$ fois –

- l'association de **for** et de **range** rend très facile de faire une opération  $n$  fois :

```
n = int(input('rentrez un nombre : '))
for i in range(n):
    print('bonjour')
```

- **i** prend les valeurs du **range**, à savoir **0,1,...,n-1**
- Autre exemple : les statistiques sur la somme de deux dés

```
from random import randint
stats = [0]*13
for i in range(1000):
    stats[randint(1,6)+randint(1,6)] += 1
print(stats)
```

## – Utiliser la suite des valeurs d'un range –

- | Afficher les nombres de 1 à  $n$ :

```
n = int(input('rentrez un nombre : '))
for i in range(n):
    print(i)
```

- Compte à rebours:

```
from time import sleep
for i in range(5,0,-1):
    print(str(i) + '... ')
    sleep(1)
print('BOOOM')
```

## – Itérer sur une liste –

- Afficher un à un les éléments d'une liste:

```
lst = [3,5,6,14,-6,121]
for x in lst:
    print(x)
```

- Changement de couleur:

```
lst = ['red','blue','green','gray','black']
for couleur in lst:
    effaceTout()
    cerclePlein(200,200,100,couleur)
    cercle(200,200,100)
    miseAJour()
    sleep(1)
attenteClic()
```

### – Itérer sur les indices d'une liste –

- Si on a besoin des indices lors du parcours d'un itérable **iterable**, on peut utiliser `range(len(iterable))`, vu que les indices vont de **0 à len(iterable)-1**

```
lst = ['bon','jour','bonjour']
for i in range(len(lst)):
    print(i,lst[i])
```

## – Itérer sur une chaîne –

- Compter le nombre de voyelles :

```
s = input('texte : ')
nbrVoyelles = 0
for a in s.lower():
    if a in ['a','e','i','o','u','y']:
        nbrVoyelles += 1
print('il y a',nbrVoyelles,'voyelles')
```

- Jeu du pendu (extrait) :

```
motPendu = ''
for a in mot:
    if a in proposes: # c'est une lettre proposee?
        motPendu += a
    else:
        motPendu += ' - '
```

## – continue et break –

- On peut utiliser les instructions **continue** et **break** avec les boucles **for** :
  - **continue** reprend au **for** en passant à l'élément suivant de l'itérable
  - **break** interrompt la boucle

```
n = int(input('nombre : '))
for i in range(2,n):
    if n % i == 0:
        print(n,'n\'est pas premier')
        print('il est divisible par',i)
        break
```

## – Conclusion sur la boucle for –

- On peut **toujours faire** une boucle **while** à la place ... c'est ce qu'on a fait jusqu'ici
- L'instruction **for** est plus**compacte**, plus **lisible**, et donc souvent meilleure quand elle est utilisable
- Elle n'est typiquement **pas adaptée** quand on ne sait pas au début de la boucle combien de fois on va l'effectuer (ex: deviner un nombre)

# Les fonctions

## Généralités sur les fonctions et les modules sous Python

1. Meilleure organisation du programme (regrouper les tâches par blocs : lisibilité → maintenance)
2. Eviter la redondance (pas de copier/coller → maintenance, meilleure réutilisation du code)
3. Possibilité de partager les fonctions (via des modules)
4. Le programme principal doit être le plus simple possible

Pourquoi créer des fonctions ?

Qu'est-ce qu'un module sous Python ?

1. Module = fichier « `.py` »
2. On peut regrouper dans un module les fonctions traitant des problèmes de même nature ou manipulant le même type d'objet
3. Pour charger les fonctions d'un module dans un autre module / programme principal, on utilise la commande `import nom_du_module`
4. Les fonctions importées sont chargées en mémoire. Si collision de noms, les plus récentes écrasent les anciennes.

## – Présentation générale –

```
def estPremier(n):
    if n < 2:
        return False
    for i in range(2,n):
        if n % i == 0:
            return False
    return True
```

- Une **fonction** est un **bloc d'instruction réutilisable**
- Cela permet d'écrire le code une seule fois pour réaliser une même tâche répétée:
  - Une fois **bien testée**, on s'en ressert **autant qu'on veut**
  - **Maintenance** à effectuer à **un seul endroit**
  - On peut mettre les fonctions **dans un module** pour **les réutiliser**
- Idée **fondamentale** en programmation : découper un programme en sous-tâches pour gagner en **lisibilité** et en **robustesse**.

## – Définir une fonction –

```
def nomFonction():
    instruction 1 de la fonction
    ...
    fin du bloc de la fonction
```

- **Important :** lors de la définition d'une fonction, le code **n'est pas exécuté**

## – Appeler une fonction –

- A tout moment dans le **programme** ou dans une **fonction** on peut appeler la fonction avec la commande **nomFonction()**

```
for i in range(2,100):
    if estPremier(i):
        print(i)
```

## – Premier exemple –

```
def appel():
    print('-' * 5, 'appel', '-' * 5)

print('bonjour')
appel()
n = int(input('nombre = '))
for i in range(n):
    appel()
```

- A chaque fois qu'on utilise l'instruction `appel()` le programme **interrompt le flot normal d'instructions** pour aller effectuer les instructions d'`appel()`
- Une fois les instructions d'`appel()` effectuées, le programme **reprend là où il en était**

## Appels des fonctions

Passage de paramètres par position

```
print(petit(10, 12))
```

Passer les paramètres selon les positions attendues  
La fonction renvoie 10

Passage par nom. Le mode de passage que je préconise, d'autant plus que les paramètres ne sont pas typés.

```
print(petit(a=10, b=12))
```

Aucune confusion possible → 10

```
print(petit(b=12, a=10))
```

Aucune confusion possible → 10

En revanche...

```
print(petit(12, 10))
```

Sans instructions spécifiques, le passage par position prévaut  
La fonction renvoie → 0

## – Fonction avec paramètres –

```
def affiche ( s ):
    print ('*' * ( len(s) + 4 ))
    print ('*' + s + '*' )
    print ('*' * ( len(s) + 4 ))

affiche ('bonjour')
texte = input()
affiche (texte)
```

- Une fonction peut avoir un ou plusieurs **paramètres**
- Ils sont nommés **entre parenthèses** dans la définition de la fonction
- Lorsque l'on appelle la fonction, il faut **passer les paramètres** (le bon nombre) entre parenthèses

## – L'instruction **return** –

- l'instruction **return** x interrompt l'exécution de la fonction et retourne la valeur x
- x peut être de n'importe quel type
- On récupère la valeur retournée normalement, par exemple par une affectation:  
`y = maFonction(x)`
- On peut aussi l'utiliser dans une expression où elle est évaluée:  
`y = maFonction(x) + 3`  
`print(maFonction(x))`
- Par défaut, si il n'y a pas de **return** ou si on met **return** simplement sans argument après la fonction retourne **None**

## – Exemple de return –

```
def minimum(lst):
    if len(lst) == 0:
        return
    mini = lst[0] #on initialise à lst[0]
    for x in lst:
        if x < mini:
            mini = x
    return mini
```

- Le premier **return** n'a pas d'argument, il retourne **None** et arrête la fonction. Le programme reprend là où il en était.
- Le second **return** renvoie le résultat (flottant) du calcul

## – Portée des variables –



Attention il y a des subtilités ici, à bien travailler.

```
def f(n):
    n = n + 1
x = 3
f(x)
print('x vaut', x)
```

- Le résultat est **x vaut3**
- Ce qui se passe :
  - à l'appel de la fonction, la valeur du paramètre de **f** est affecté au **n** de la **définition def**
  - donc **n** vaut **3**
  - dans la fonction, **n** est augmenté de 1
  - **x** n'a pas changé
  - ... d'ailleurs **n** n'existe pas dans le corps du programme

## – Portée des variables –



Attention il y a des subtilités ici, à bien travailler.

```
def f(n):
    n = n + 1
n = 3
f(n)
print('n vaut', n)
```

- Le résultat est **encore n vaut 3 !**
- Ce qui se passe :
  - à l'appel de la fonction, la valeur du paramètre de **f** est affecté au **n** de la **définition def**
  - Ce **n'est pas** le même **n**
  - Il y a le **n** principal, et le **n** de **f** qu'on va noter  **$n_f$**
  - **$n_f$**  prend la valeur de **n** à l'appel de **f** et est incrémenté de 1 dans la fonction. **n** ne change pas.

## – Portée des variables –



Attention il y a des subtilités ici, à bien travailler.

```
def f(x):
    n = 1
n = 3
f(n)
print('n vaut', n)
```

- Le résultat est **toujours n vaut 3 !**
- Ce qui se passe :
  - à l'appel de la fonction, la valeur du paramètre de **f** est affecté au **x** de la **définition** de **f**
  - L'affectation dans la fonction crée une variable **locale** à **f**, notons-là **n<sub>f</sub>**
  - **n<sub>f</sub>** prend la valeur de 1 et le **n** principal ne change pas.

## – Porté des variables –

- les paramètres de la définition de la fonction sont des **variables locales**, propres à la fonction
- les variables affectées dans la fonction sont des **variables locales**, propres à la fonction
- ces variables locales existent **pendant l'exécution de la fonction** et n'existent plus après
- les variables affectées dans le corps du programme (hors fonctions) sont des **variables globales**
- les **variables globales** sont lisibles dans tout le programme
- les **variables globales** ne sont pas modifiables dans une fonction
- (si on affecte une **variable globale** dans une fonction, on crée une **variable locale** avec le même nom)
- pour modifier une variable globale **x** dans une fonction, il faut la déclarer avec le mot clé **global**

## – Exemple de portée –

```
def f(n):
    global k
    i = n
    k = 0
    print(i, j, k)
i = 2
j = 4
k = 6
f(44)
print(i, j, k)
```

- Dans le corps de la fonction **n** et **i** sont des variables locales
- **k** est une variable globale modifiable
- **j** est visible en tant que variable globale

# **Utilisation des modules**

## Modules

## Modules standards

Voir la liste complète sur



<https://docs.python.org/3/library/>

## Principe des Modules - Les modules standards de Python

- Un module est un fichier « **.py** » contenant un ensemble de variables, fonctions et classes que l'on peut importer et utiliser dans le programme principal (ou dans d'autres modules).
  - Le mot clé **import** permet d'importer un module
  - C'est un pas supplémentaire vers la modularité : un module maximise la réutilisation et facilite le partage du code
- 
- Des modules standards prêts à l'emploi sont livrés avec la distribution Python. Ex. random, math, os, hashlib, etc.
  - Ils sont visibles dans le répertoire « Lib » de Python

## – Modules –

- un **module** est un ensemble de fonctions déjà programmées, prêtes à être utilisées
- il existe des **blue modules de base** en python (`math` par exemple), mais vous pouvez en récupérer sur internet ou en créer vous-même
- utilisé pour organiser les programmes (on en reparlera)

Il y a deux façons d'utiliser une fonction `toto()` d'un module `monModule`:

- `from monModule import toto` : on utilise alors normalement la fonction
- `import monModule` : la fonction s'appelle `monModule.toto()`
- on peut aussi utiliser `from monModule import *` pour charger toutes les fonctions du module `monModule`, sauf celles qui commencent par

## Exemple d'utilisation de modules standards

```
# -*- coding: utf -*-
# importer les modules
#math et random
import math, random

#générer un nom réel
#compris entre 0 et 1
random.seed(None)
value = random.random()

#calculer le carré de
#son logarithme
logv = math.log(value)
abslog = math.pow(logv, 2.0)

#affichage
print(abslog)
```

Si plusieurs modules à importer, on les met à la suite en les séparant par « , »

Préfixer la fonction  
à utiliser par le  
nom du module

## Autres utilisations possibles

```
#définition d'alias
import math as m, random as r

#utilisation de l'alias
r.seed(None)
value = r.random()
logv = m.log(value)
abslog = m.pow(logv,2.0)
```

L'alias permet d'utiliser des noms plus courts dans le programme.

```
#importer le contenu
#des modules
from math import log, pow
from random import seed, random

#utilisation directe
seed(None)
value = random()
logv = log(value)
abslog = pow(logv,2.0)
```

Cette écriture permet de désigner nommément les fonctions à importer.

Elle nous épargne le préfixe lors de l'appel des fonctions. Mais est-ce vraiment une bonne idée ?

N.B.: Avec « \* », nous les importons toutes (ex. `from math import *`). Là non plus pas besoin de préfixe par la suite.

## – Un exemple –

```
from math import *
print('pi = ', pi)
print(cos(pi/2))
```

```
Import math
print('pi = ', pi)
print(cos(pi/2))
```

## – Quelques modules utiles –

- **random** : des fonctions pour faire des tirages au sort
- **math** : les fonctions et les constantes mathématiques usuelles comme  $\exp$ ,  $\cos$ ,  $\uparrow$ , ...
- **time** : pour mesurer le temps, connaître l'heure, ou attendre un certain temps
- **tkinter** : pour les interfaces graphiques (button, zone de texte, label, ...)

# **Introduction au langage Python**

**Cours 3/4**

**Structures de données  
avancées**

**Janvier 2020**

## – tuple –

- Les **tuple** sont l'équivalent de la notion mathématique de  $n$ -uplets
- Déclaration **x = (4,3,1)** crée un tuple avec 3 entiers
- On peut aussi directement écrire **x = 4,3,1**
- Pour faire un **tuple** avec **un seul élément**, il faut utiliser une virgule : **x = (4,)**, sinon **x** est un **int** qui vaut 4
- On accède au  $i$ -ème élément d'un **tuple** comme pour les listes ou les chaînes : **x[i]**, où les indices commencent à 0
- La longueur d'un tuple est renvoyée par la fonction **len()**
- On peut changer un **itérable** en tuple à l'aide de la fonction
  - **tuple()** : **tuple(range(5))** → **(0,1,2,3,4)**
- On peut concaténer deux **tuple** avec +

## Création des tuples et accès aux données

```
#définition d'un tuple  
t1 = (2,6,8,10,15,26)  
print(t1)  
  
#taille du tuple  
print(len(t1))  
  
#accès indicé  
a = t1[0]  
print(a)  
  
#modification ?  
t1[2] = 3  
  
#plage d'indices  
b = t1[2:5]  
print(b)  
  
#autre plage  
c = t1[:4]  
print(c)  
  
#indication négatif  
d = t1[-1]  
print(d)  
  
#indication négatif  
e = t1[-3:]  
print(e)
```

Les () sont importantes pour indiquer qu'il s'agit d'un tuple, « , » sépare les éléments.

(2,6,8,10,15,26)

6 éléments

1<sup>er</sup> élément, les indices vont de 0 à len(t1)-1

Remarque : a n'est pas un tuple

ERREUR

Attention : on récupère à partir du n°2 (inclus) au n°5 (non-inclus) c.-à-d. les indices 2, 3, 4

Résultat : b est un tuple avec (8,10,15)

Les 4 premiers éléments c.-à-d. les indices 0, 1, 2, 3 : nous obtenons le tuple (2, 6, 8, 10).

Le 1<sup>er</sup> élément à partir de la fin : 26

Les 3 derniers éléments : (10,15,26)

## Plus loin avec les tuples

```
#concaténation  
t2 = (7, 9, 31)  
t3 = t1 + t2  
print(t3)
```

(2,6,8,10,15,26,7,9,31)

```
#réplication  
t4 = 2 * t2  
print(t4)
```

(7,9,31,7,9,31)

```
#tuples d'objets hétérogènes  
v1 = (3, 6, "toto", True, 34.1)  
print(v1)
```

Ca ne pose absolument aucun problème.

```
#tuple de tuples  
x = ((2,3,5), (6,7,9), (1,8))  
print(x)
```

Sorte de tableau à 2 dimensions

```
#accès indicé  
print(x[2][1])
```

→ 8



x[0] → (2,3,5)  
x[1] → (6,7,9)  
x[2] → (1,8)

Organisation de la structure

```
#accès aux tailles  
print(len(x))  
print(len(x[2]))
```

→ 3

→ 2

3 éléments sur la 1<sup>ère</sup> dimension

2 éléments dans le tuple référencé par x[2]

## – tuple vs list –



Même s'ils se ressemblent, **tuple** et **list** sont des structures complètement différentes

- La principale différence c'est que:
  - Une **list** est **modifiable**
  - Un **tuple n'est pas modifiable**
- Si **t** est un tuple, **t[0] = 3** produit une erreur car **t** n'est pas modifiable
- Il n'y a pas de **append()** pour les **tuple**
- ...
- En fait, un **tuple** ressemble plus à une chaîne qu'à une liste

## Bilan sur les tuples

- Type ‘tuple’ cf. appel de la fonction type()
- Collection d’objets de types hétérogènes
- Taille et contenu fixé lors de l’écriture du programme
- Impossible de modifier : objet non mutable
- La variable de type tuple est en réalité une référence (pointeur de pointeur)
- Bénéficie du mécanisme de ramasse miettes
- Accès indicé, plage d’indices possible, indices négatifs possibles aussi
- Structures complexes avec tuple de tuples, et même plus – à voir plus tard

## – Set –

- Set en anglais signifie ensemble
- La structure set permet de gérer efficacement un ensemble de donnée
- Comme c'est un ensemble, chaque élément ne peut y être violet qu'une seule fois
- Comme c'est un ensemble, l'ordre ne compte pas
- Le mécanisme utilisé pour que cette structure soit efficace fait que les éléments d'un set doivent être non modifiables
- Un set est un objet modifiable



On ne peut mettre dans un set que des objets non modifiables, donc pas de liste, pas de set et pas de dictionnaire

## Propriétés des ensembles

- Un ensemble est une collection non ordonnée d'éléments.
- Chaque élément est unique (pas de doublons) et doit être immuable (ce qui ne peut pas être changé).
  - Un ensemble ne peut pas avoir un élément mutable, comme une liste, un ensemble ou un dictionnaire.
- Un ensemble diffère d'une liste des manières suivantes:
  - Chaque élément à l'intérieur d'un ensemble doit être unique.
  - Les éléments d'un ensemble ne sont stockés dans aucun ordre particulier.
- Les ensembles peuvent être utilisés pour effectuer des opérations d'ensembles mathématiques telles que l'union, l'intersection, la différence symétrique, etc.
- Un ensemble est créé en plaçant tous les éléments entre accolades {}, séparés par une virgule ou en utilisant la fonction intégrée set().
- Les éléments peuvent être de types différents (int, float, tuple, string, etc.).

## – Opération sur les Set –

Dans le tableau, **s** et **t** sont des **set** et **l** est un **itérable**:

<b>set()</b>	crée un set vide	<b>len(s)</b>	longueur du set <b>s</b>
<b>x in s</b>	teste si $x \in s$	<b>x not in s</b>	teste si $x \notin S$
<b>s &lt;= t</b>	teste si $s \subseteq t$	<b>s   t</b>	retourne $s \cup t$
<b>s &amp; t</b>	retourne $s \cap t$	<b>s ^ t</b>	retourne $s \Delta t$
<b>s.add(x)</b>	ajoute $x$ dans $s$	<b>s.remove(x)</b>	retire $x$ de $s$
<b>s.pop()</b>	retourne et enlève un élément de $S$	<b>s.discard(x)</b>	comme <b>remove</b> , mais pas d'erreur si $x \notin S$
<b>set(l)</b>	crée un set avec <b>l</b>	<b>s==t</b>	teste l'égalité

- les **opérations en bleu** dans la table **modifient** le set **s**

### – Test d'efficacité –

- On crée une **list** avec 10 000 entiers
- On teste si les 10 000 entiers sont dedans
- On fait la même chose avec un **set**
- On compare le temps d'exécution avec le module **time**

## – Dictionnaires –

- Les **dictionnaires** permettent d'implanter de façon très efficace des fonctions (partielles)
- Dans certains langages, ils sont appelés des **tableaux associatifs**
- Cela permet d'associer à une **clé** une **valeur**
- Exemple d'utilisation :

```
D = {} # dictionnaire vide
D['toto'] = 4 # associe 4 a la cle 'toto'
D['titi'] = 6
print(D['toto']) # affiche 4
D['toto'] = 'bonjour' # remplace 4
print(D['toto']) # affiche 'bonjour'
print(D)
```



La clé doit être un élément **non modifiable**

## Dictionnaire- Le type dict

```
#définition d'un dictionnaire  
d1 = {'Pierre':17, 'Paul':15, 'Jacques':16}  
print(d1)  
#ou  
print(d1.items())  
  
#nombre d'éléments  
print(len(d1))  
  
#liste des clés  
print(d1.keys())  
  
#liste des valeurs  
print(d1.values())  
  
#accès à une valeur par clé  
print(d1['Paul'])  
#ou  
print(d1.get('Paul'))  
  
#si clé n'existe  
pas  
print(d1['Pipa'])
```

Noter le rôle de {}, de « : » et « , »

→ 3 éléments

[‘Paul’, ‘Jacques’, ‘Pierre’]

[15, 16, 17]

→ 15

→ 15

→ ERREUR

Dictionnaire : collection non-ordonnée (non indicée) d'objets (simples ou évolués) s'appuyant sur le mécanisme associatif « clé – valeur ».

### Remarques :

- 1) `d1.clear()` vide le dictionnaire
- 2) `d1` est une référence, `d1.copy()` permet de copier le contenu.

## Dictionnaire – Modifications, ajouts et suppressions

```
#modification  
d1['Jacques'] = 18    {'Pierre':17,'Paul':15,'Jacques':16} → {'Pierre':17,'Paul':15,'Jacques':18}  
print(d1)  
  
#ajouter un élément  
d1['Henri'] = 22      {'Pierre':17,'Paul':15,'Jacques':18, 'Henri':22}  
print(d1)              Ajout par définition d'une nouvelle paire « clé – valeur ». N.B. : Si 'Henri' existe déjà, son ancienne valeur sera écrasée.  
  
#ajout d'un bloc d'éléments  
d1.update({'Monica':36,'Bill':49})  
print(d1)              {'Pierre':17,'Paul':15,'Jacques':18,'Henri':22,'Monica':36,'Bill':49}  
  
#déetecter présence clé  
test = 'Pierre' in d1  
print(test)            → True  
  
#suppression par clé  
del d1['Monica']  
print(d1)              {'Pierre':17,'Paul':15,'Jacques':18,'Henri':22,'Bill':49}
```

## Plus loin avec les clés

Les clés ne sont pas forcément des chaînes de caractères.  
L'outil est très souple mais, attention, autant de liberté  
peut être aussi préjudiciable. Il faut être très rigoureux.

```
#autre type de clé
d2 = {('Pierre',56):['Directeur',1253,True],('Paul',55):['Employé',100,False]}
print(d2.keys())
print(d2.values())
```

Dans cet exemple :

- **clé** est un tuple ;
- **valeur** est une liste.

D'autres types sont possibles (ex. instances de classes – à voir plus loin).



## Exemple

```
dico_exemple.py - D:/_Travaux/university/Cours_... - □ X
File Edit Format Run Options Window Help
# -*- coding: utf -*-
#nombre d'items
n = int(input("Nombre d'items : "))

#dictionnaire vide
dico = {}

#saisie
for i in range(0,n):
    cle = input("Clé : ")
    valeur = float(input("Valeur : "))
    dico[cle] = valeur

#liste des clés
print(dico.keys())

#liste des valeurs
print(dico.values())

#affichage par paire
for (k,v) in dico.items():
    print(k,v)

#somme des valeurs
s = 0
for v in dico.values():
    s = s + v
print(s)
```

Accès indicé malaisé (**très**), on a intérêt à passer par un itérateur pour passer en revue le dictionnaire.

Exemple :

Kate	15.0
Pipa	23.5
William	10.7

49.2

## – Opération sur les dictionnaires –

Dans le tableau suivant D est un dictionnaire, x est une clé et y est une valeur :

{ }	dictionnaire vide	<b>len(D)</b>	nombre de clés
<b>D[x] = y</b>	D[x] vaut y	<b>D[x]</b>	retourne la valeur de x
<b>del D[x]</b>	x n'a plus de valeur	<b>x in D</b>	teste si x est une clé
<b>D.keys()</b>	la liste de clés	<b>D.values()</b>	la liste des valeurs
<b>D.items()</b>	la liste des couples (clé,valeur)		

## – Modifiables et non-modifiables –

On a vu des structures **non-modifiables**:

- booléens, entiers, caractères, flottants
- chaînes de caractère
- tuple

Et des structures **modifiables** :

- listes
- ensembles
- dictionnaires

Il peut être utile de passer d'un type à l'autre. On peut par exemple utiliser la fonction **tuple()** pour transformer une liste (**modifiable**) en un tuple (**non-modifiable**)

## – Affectation multiple –

- On peut affecter **simultanément** plusieurs variables avec la syntaxe **x,y,z = iterable**
- Cela ne fonctionne que s'il y a le **même nombre** d'éléments à gauche que dans l'itérable **x,y,z = [5,6,8]**
- Comme on peut **omettre les parenthèses** lors de l'écriture d'un tuple, on peut utiliser **x,y,z = 3,6,8**
- On peut même écrire **x,y = y,x** ce qui **échange les deux variables!**
- Il est possible d'utiliser pour signifier des positions qui ne nous intéressent pas

**x,\_,\_,y = range(4)**

- On peut s'en servir dans **toutes les situations**, par exemple

```
for key , value in D.items():
    print('clé=' ,key , 'valeur=' ,value )
```

# Compléments sur les fonctions

## – Commentaire de fonction –

- | Après l'entête de la fonction, on peut mettre un **descriptif de la fonction** directement dans une chaîne de caractères

```
def pgcd(a,b):
    'calcule le pgcd de a et de b'
    while b != 0:
        a, b = b, a % b
    return a
```

- On accède à la description avec la fonction **help** dans un terminal Python
- Certains éditeurs Python comme **IDLE3** font apparaître la description des fonctions
- Il faut prendre l'habitude de **mettre une description pour toute les fonctions importantes.**

## – Paramètres par défaut –

- On peut **spécifier des valeurs par défauts** dans une fonction

```
def f (x ,y=4 ,z=5):  
    return x + y + z
```

- Si les champs considérés ne sont pas donnés lors de l'appel à la fonction, ils prennent la valeur par défaut:

f(2,2,2) ! 6

f(2,3) ! f(2,3,5) ! 10

f(2) ! f(2,4,5) ! 11

f() ! **erreur**



Il **ne faut pas** mettre des valeurs modifiables comme **valeurs par défaut**, mais vous pouvez mettre des **tuple, string, ...**

## Paramètres par défaut

### – Paramètres par défaut –

- Affecter des valeurs aux paramètres dès la définition de la fonction
- Si l'utilisateur omet le paramètre lors de l'appel, cette valeur est utilisée
- Si l'utilisateur spécifie une valeur, c'est bien cette dernière qui est utilisée
- Les paramètres avec valeur par défaut doivent être regroupés en dernière position dans la liste des paramètres

## Exemple

```
def ecart(a,b,epsilon = 0.1):  
    d = math.fabs(a - b)  
    if (d < epsilon):  
        d = 0  
    return d  
  
ecart(a=12.2, b=11.9, epsilon = 1) #renvoie 0  
ecart(a=12.2, b=11.9) #renvoie 0.3
```

La valeur utilisée est epsilon = 0.1 dans ce cas

## – Paramètres modifiables –

Rappel :

```
def f(x):
    x = x + 1
n = 3
f(n)
print(n)
```

- le “x” de la fonction `f` est une variable locale de `f`, le “n” global n’est donc pas changé lors de l’appel à la fonction : cela affiche 3

Avec une liste :

```
def ajoute(L,x):
    L.append(x)
lst = [4,5]
ajoute(lst,7)
print(lst)
```

## – Paramètres modifiables (suite) –

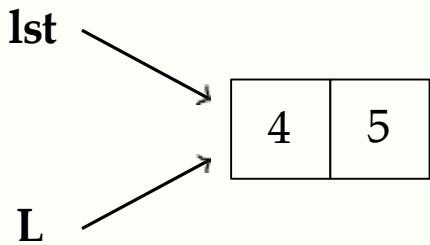
```
def f(x):
    x = x + 1 # x est incrémenté
n = 3
f(n)      # x de f prend la valeur de n
print(n)
```

```
def ajoute(L,x):
    L.append(x) # on modifie L en ajoutant x
lst = [4,5]
ajoute(lst,7) # L prend la valeur lst
print(lst) # affiche [4,5,7]
```

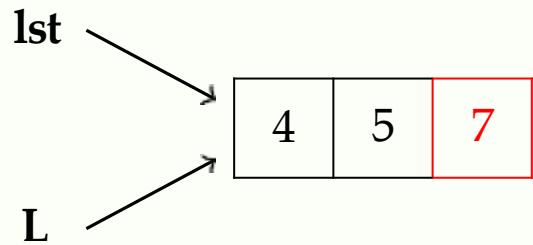
## – Paramètres modifiables (suite) –

```
def ajoute(L,x):
    L.append(x) # on modifie L en ajoutant x
lst = [4,5]
ajoute(lst,7) # L prend la valeur lst
print(lst) # affiche [4,5,7]
```

ajoute(**lst,7**) ! L,x = **lst,7**

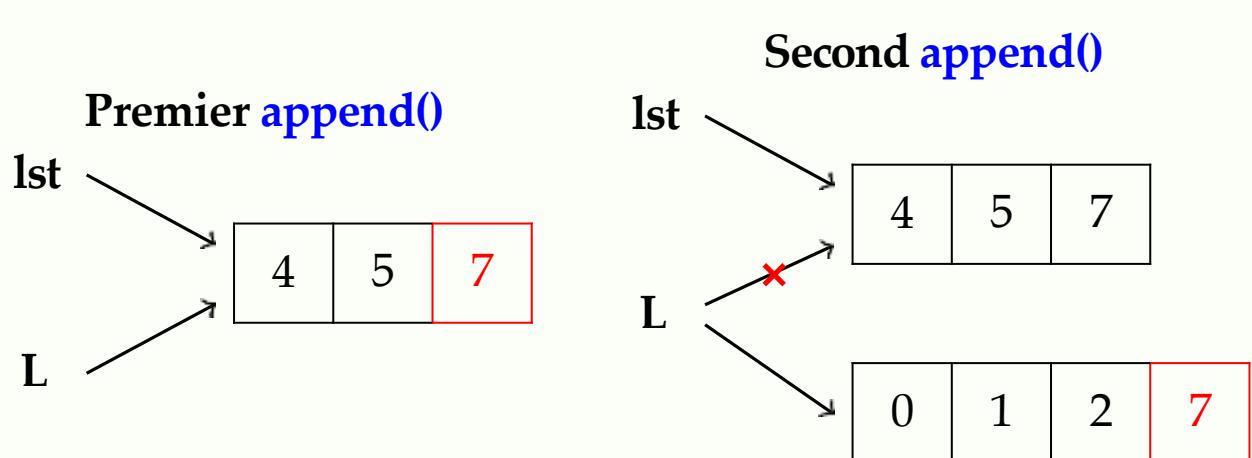


**L.append(x)**



## – Paramètres modifiables (fin) –

```
def f(L, x):
    L.append(x)
    L = list(range(3))
    L.append(x)
lst = [4, 5]
f(lst, 7)
```



## Fonction renvoyant plusieurs valeurs (1)

Renvoyer plusieurs valeurs avec return

```
#écriture de la fonction
def extreme(a,b):
    if (a < b):
        return a,b
    else:
        return b,a

#appel
x = 10
y = 15
vmin,vmax = extreme(x,y)
print(vmin,vmax)
```

vmin =10 et vmax=15

return peut envoyer plusieurs valeurs simultanément.  
La récupération passe par une affectation multiple.

Remarque: Que se passe-t-il si nous ne mettons qu'une variable dans la partie gauche de l'affectation ?

```
#ou autre appel
v = extreme(x,y)
print(v) ←
#quel type pour v ?
print(type(v))
```

v = (10, 15)

<class 'tuple'>

v est un « tuple », une collection de valeurs, à voir plus tard.

## Utilisation des listes et des dictionnaires

### Fonction renvoyant plusieurs valeurs (2)

Nous pouvons aussi passer par une structure intermédiaire telle que la liste ou le dictionnaire d'objets. Les objets peuvent être de type différent, au final l'outil est très souple. (nous verrons plus en détail les listes et les dictionnaires plus loin)

```
#écriture de la fonction
def extreme_liste(a,b):
    if (a < b):
        return [a,b]
    else:
        return [b,a]

#appel
x = 10
y = 15
res = extreme_liste(x,y)
print(res[0])
```

```
#écriture de la fonction
def extreme_dico(a,b):
    if (a < b):
        return {'mini' : a, 'maxi' : b}
    else:
        return {'mini' : b, 'maxi' : a}

#appel
x = 10
y = 15
res = extreme_dico(x,y)
print(res['mini'])
```



- Les deux fonctions renvoient deux objets différents
- Notez l'accès à la valeur minimale selon le type de l'objet

## – Autre exemple : l'alphabet –

```
def ajouteLettres(u,D):
    for x in u:
        D.add(x)

A = set()
while True:
    s = input('mot = ')
    if s == 'stop':
        break
    ajouteLettres(s,A)
print('alphabet=',A)
```

## – Fonctions en argument –

- | On peut passer une **fonction en argument** d'une autre fonction

```
def filtre(L, f):
    R = []
    for x in L:
        if f(x):
            R.append(x)
    return R

def estPair(n):
    return n % 2 == 0

def toto(n):
    return n % 3 == 0

lst = filter(range(10), estPair)
print(lst)
print(filter(range(20), toto))
```

## – Tri avec plusieurs fonctions de comparaison –

```
def triBulle(T, plusGrand):
    for i in range(len(T)-1, 0, -1):
        for j in range(i):
            if plusGrand(T[j], T[j+1]):
                T[j], T[j+1] = T[j+1], T[j]

def superieurDebut(u, v):
    return u[0] > v[0]

def superieurFin(u, v):
    return u[len(u)-1] > v[len(v)-1]

def usuel(x, y):
    return x > y
```

## -Fonctions lambda -

- Il peut être utile de créer une fonction à la volée pour la passer en **paramètre**
- On peut le faire grâce au mot clé **lambda**, la syntaxe est

**lambda x : expression(x)**

qui est une fonction anonyme qui est l'équivalent de  
**def f(x):**

**return expression(x)**

```
s = "ceci est le dernier cours de python"
lst = s.split()
print(sorted(lst, key=lambda x:x[0]))
```

## – (illusion de) retourner plusieurs valeurs –

- Comme une fonction peut retourner un **tuple**, on peut s'en servir pour retourner plusieurs valeurs

```
def divEuclidienne(a,b):
    'retourne quotient et reste'
    return a // b, a % b

q, r = divEuclidienne(14,4)
print('quotient=' , q, ', reste=' , r)
```

# **Introduction au langage Python**

**Cours 4/4**

**Les fichiers**

**Janvier 2020**

# Lecture / écriture dans un fichier

## Fichiers ?

**Fichier** - collection d'informations stockées sur une mémoire de masse (non volatile, capacité plus importante que la mémoire vive)

**Types de fichiers** – Ils se distinguent par...

1. **Organisation des données** : structuré, les informations sont organisées d'une manière particulière qu'il faut respecter (ex. fichiers d'enregistrements) ; non-structuré, les informations sont alignées à la suite (ex. fichier texte)
2. **Mode d'accès aux données** : accès indicé, utilisable comme un tableau ; accès séquentiel, lecture ou écriture pas à pas (ex. ligne par ligne dans un fichier texte)



La catégorisation n'est pas toujours très tranchée, un fichier XML par ex. est un fichier texte (manipulable avec un éditeur de texte) mais qui obéit à une organisation structurée

## – L'instruction `join()` –

- l'instruction `join()` est une instruction (méthode) de chaîne de caractères
- On l'utilise de la façon suivante :  
`s.join(it)`  
où `s` est une chaîne de caractères et `it` est un itérable contenant des chaînes de caractères.
- le résultat est une chaîne qui contient les mots de `it` reliés par `s`
- `'.'.join(['ab','cd','efg'])` !    'ab:cd:efg'

## – Ouverture et fermeture d'un fichier –

- On peut instancier une variable de type **fichier**, qui va permettre de faire des opérations sur les fichiers présents sur l'ordinateur
- Pour **ouvrir** un fichier en python, on utilise la commande :  
**f = open(chemin,mode),**  
où **f** est la variable qu'on utilisera pour accéder au fichier, **chemin** est le nom du fichier (éventuellement avec le chemin pour le trouver **'./toto.txt'**) et **mode** est le mode d'utilisation du fichier dans le programme
- Il existe de nombreux modes d'accès aux fichiers, voilà les trois plus communs :
  - **'r'** : mode lecture seulement, c'est le mode par défaut
  - **'w'** : mode écriture, le fichier est créé s'il n'existe pas, sinon il est effacé pour pouvoir y écrire
  - **'a'** : mode ajout, c'est un mode écriture à partir de la fin du fichier
- Pour **fermer** un fichier : **f.close()**

## – Les objets file –

- C'est un objet **modifiable** : si on le fait évoluer dans une fonction, il évolue globalement
- Il connaît le fichier
- Il a une **position courante** dans le fichier, qui est modifiée au fur et à mesure qu'on lit ou écrit dans le fichier

## – Lecture dans un fichier –

- Il faut que le fichier soit **ouvert en lecture**
- On peut lire une ligne de **f** avec l'instruction **f.readline()**
- Cela **déplace la position courante** à la ligne suivante
- Donc on peut répéter l'appel à **f.readline()** pour lire toutes les lignes une à une
- Quand il n'y a plus rien à lire, **f.readline()** retourne la chaîne vide “”

```
f = open('filtre.py')
ligne = None
while ligne != '':
    ligne = f.readline()
    print(l)
f.close()
```

## – Solution alternative –

- Une file **f** peut aussi être vue comme une **structure itérable** de ses lignes
- Cela permet de très facilement lire les lignes de **f**

```
f = open('iterable.py')
for ligne in f:
    print(ligne)
f.close()
```



Que l'on utilise **readline()** ou le format d'itérable, les lignes rentrées conservent le caractère '**\n**' à la fin. On peut l'enlever avec l'instruction **ligne = ligne[:-1]** (cf dernier cours)

## – Ecriture –

- Pour écrire dans un fichier, il faut l'ouvrir en **écriture 'w'** ou en **ajout 'a'**
- Pour écrire la chaîne **s** dans le fichier **f**, on utilise l'instruction **f.write(s)**
- Attention, contrairement à **print( )**, cela ne rajoute pas un saut de ligne à la fin

```
f = open('tmp','w')
for i in range(10):
    f.write('ligne '+str(i)+'\n')
f.close()
```

## – Exemples –

- Lister les palindromes en français
- Créer un nouveau fichier sans les accents et sans les ç
- Lister les palindromes du nouveau fichier
- Le jeu du pendu
- Recherche d'anagrammes :
  - Un ensemble de lettres (avec répétitions) est vu comme un tuple ordonné (on utilise la fonction `lst.sort()` qui tri la liste `lst`)
  - On stocke les anagrammes sous forme d'un dictionnaire où les clés sont les tuples ordonnés ci-dessus, et les valeurs l'ensemble des mots qui utilisent ces lettres

## FICHIER TEXTE

Accès séquentiel, non structuré

## Lecture en bloc avec `read()`

```
# -*- coding: utf -*-
#ouverture en lecture
f = open("voitures.txt", "r")

#lecture
s = f.read()

#affichage
print("** contenu de s **")
print(s)
print("** fin contenu **")

#information sur s
print("type de s : ", type(s))
print("longueur de s : ", len(s))

#fermeture
f.close()
```

Fichier texte à lire :  
« voiture.txt »

```
megane
clio
twingo
safrane
laguna
vel satis
```

- `open()` permet d'ouvrir un fichier, en lecture ici avec l'option « `r` ». La fonction renvoie un objet de type fichier stocké dans `f`
- le curseur de fichier est placé sur la première ligne
- `read()` lit tout le contenu du fichier d'un bloc
- `close()` ferme le fichier (et donc le déverrouille)

`s` est de type « `str` », on se rend mieux compte en mode console:

```
>>> s
'megane\nclio\n\twingo\n\tsafrane\n\tlaguna\n\tvel satis'
```

`\n` représente le caractère spécial « saut de ligne » (line feed),  
c'est pour cela que `print(s)` fonctionne correctement.

```
** contenu de s **
megane
clio
twingo
safrane
laguna
vel satis
** fin contenu **
type de s : <class 'str'>
longueur de s : 43
```

## Lecture en bloc avec `readlines` (avec « s »)

```
# -*- coding: utf -*-
#ouverture en lecture
f = open("voitures.txt", "r")

#lecture
lst = f.readlines()

#affichage
print("** contenu de lst **")
print(lst)
print("** fin contenu **")

#information sur lst
print("type de s : ", type(lst))
print("longueur de s : ", len(lst))

#fermeture
f.close()
```

Le contenu du fichier est stocké dans une liste, une ligne = un élément. Le caractère `\n` est maintenu. Il n'est pas présent sur la dernière ligne de notre fichier exemple.

```
** contenu de lst **
['megane\n', 'clio\n', 'twingo\n', 'safrane\n', 'laguna\n', 'vel satis']
** fin contenu **
type de s :  <class 'list'>
longueur de s :  6
```

**Remarque : saut de ligne ou pas sur la dernière ligne du fichier**

Ouvrir le fichier dans Notepad++



**Question :** Comment savoir s'il y a un saut de ligne ou pas à la dernière ligne de notre fichier texte ?

```
voitures.txt
1 megane
2 clio
3 twingo
4 safrane
5 laguna
6 vel satis
```

La ligne n°6 est la dernière ligne du fichier,  
pas de saut ligne après « vel satis »

```
** contenu de lst **
['megane\n', 'clio\n', 'twingo\n', 'safrane\n', 'laguna\n', 'vel satis']
** fin contenu **
type de s : <class 'list'>
longueur de s : 6
```

```
voitures.txt
1 megane
2 clio
3 twingo
4 safrane
5 laguna
6 vel satis
7
```

Il y a une ligne **vide** (la n°7) après « vel satis »

```
** contenu de lst **
['megane\n', 'clio\n', 'twingo\n', 'safrane\n', 'laguna\n', 'vel satis\n']
** fin contenu **
type de s : <class 'list'>
longueur de s : 6
```



## Lecture ligne par ligne avec readline (sans « s »)

```
# -*- coding: utf -*-
#ouverture en lecture
f = open("voitures.txt","r")

#lecture ligne itérativement
while True:
    s = f.readline()
    if (s != ""):
        print(s)
    else:
        break;

#fermeture
f.close()
```



```
megane
clio
twingo
safrane
laguna
vel satis
```

- **readline()** lit la ligne courante et place le curseur sur la ligne suivante.
- la fonction renvoie la chaîne vide lorsque nous arrivons à la fin du fichier (Remarque : s'il y a une ligne blanche entre 2 véhicules, readline() renvoie le caractère « \n », ce n'est pas une chaîne vide).

Il y a une ligne vide entre chaque véhicule parce que le caractère « \n » est toujours là, print() le prend en compte [ Remarque : pour que print() n'en tienne pas compte, il faudrait faire `print(s,end="")` ]

## Lecture ligne par ligne en itérant sur l'objet fichier

```
# -*- coding: utf -*-
#ouverture en lecture
f = open("voitures.txt","r")
#lecture ligne itérativement
for s in f:
    print(s,len(s))

#fermeture
f.close()
```

C'est la forme la plus efficace – et la plus concise – pour une lecture ligne à ligne.

Le caractère `\n` est présent toujours,  
noter la longueur de la chaîne (+1 pour toutes sauf la dernière)



```
megane
7
clio
5
twingo
7
safrane
8
laguna
7
vel satis 9
```

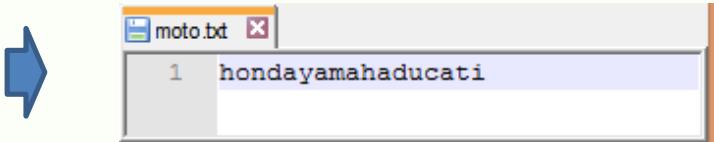
## Ecriture d'un fichier texte avec write()

```
# -*- coding: utf -*-
#ouverture en écriture
f = open("moto.txt", "w")

#écriture
f.write("honda")
f.write("yamaha")
f.write("ducati")

#fermeture
f.close()
```

- `open()` permet d'ouvrir un fichier en écriture avec l'option « `w` ». La fonction renvoie un objet de type fichier référencé par `f`
- avec « `w` », le fichier est écrasé s'il existe déjà
- `write()` permet d'écrire la chaîne de caractères



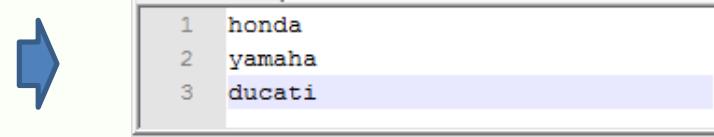
Il manque les sauts de ligne pour distinguer chaque moto

```
# -*- coding: utf -*-
#ouverture en écriture
f = open("moto.txt", "w")

#écriture
f.write("honda\n")
f.write("yamaha\n")
)
f.write("ducati")

#fermeture
f.close()
```

Nous insérons le caractère saut de ligne « `\n` » après chaque moto, sauf la dernière



## Ecriture d'un fichier texte avec writelines()

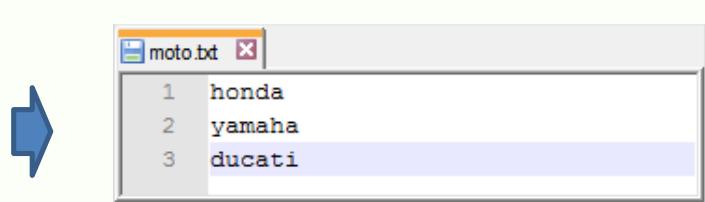
```
# -*- coding: utf -*-
#ouverture en écriture
f = open("moto.txt", "w")

#liste
lst = ["honda\n", "yamaha\n", "ducati"]

#écriture
f.writelines(lst)

#fermeture
f.close()
```

writelines() permet d'écrire directement le contenu d'une liste. Nous devons quand même insérer le caractère « \n » pour que le saut de ligne soit effectif dans le fichier.



## Ajout dans un fichier texte

```
# -*- coding: utf -*-
#ouverture en ajout
f = open("moto.txt", "a")

#ajouter un saut de
ligne f.write("\n")

#écriture
f.write("laverda")

#fermeture
f.close()
```

- `open()` avec l'option « `a` » permet d'ouvrir un fichier en mode ajout
- `write()` permet d'écrire la chaîne de caractères
- attention toujours au saut de ligne
- une ouverture en mode lecture / écriture est possible avec l'option « `r+` » mais se positionner sur telle ou telle ligne pour y modifier ou insérer des informations est compliqué.



```
moto.txt
1 honda
2 yamaha
3 ducati
4 laverda
```

# **FORMAT CSV**

Fichier structuré

## Format CSV

Un fichier CSV (Comma-separated values) est un fichier texte (!) avec une structure **tabulaire**. Ce type de fichier peut être généré à partir d'un tableur (Excel ou Calc d'Open/Libre Office). C'est un format privilégié pour le traitement statistique des données.

Excel

	A	B	C
1	age	poids	taille
2	45	65	178
3	58	89	176
4	35	85	194

Notepad++

```
1 age;poids;taille
2 45;65;178
3 58;89;176
4 35;85;194
```

**Remarques :** (1) Le passage d'une ligne à l'autre est matérialisé par un saut de ligne ; (2) ";" est utilisé comme séparateur de colonnes (paramétrable, ça peut être tabulation "\t" aussi souvent) ; (3) le point décimal dépend de la langue (problème potentiel pour les conversions) ; (4) la première ligne joue le rôle d'en-tête de colonnes souvent (nom des variables en statistique).

## Lecture du format CSV – Structure de liste

```
#ouverture en lecture
f = open("personnes.csv", "r")

#importation du module csv
import csv

#lecture - utilisation du parseur csv
lecteur = csv.reader(f, delimiter=";")

#affichage - itération sur chaque ligne
for ligne in lecteur:
    print(ligne)

#fermeture du fichier
c. close()
```

Paramétrage  
du séparateur  
de colonnes  
avec l'option  
delimiter.

Chaque ligne  
est une liste.

```
[ 'age', 'poids', 'taille']
[ '45', '65', '178']
[ '58', '89', '176']
[ '35', '85', '194']
```

### Remarques :

1. La première ligne est une observation comme une autre.
2. Toutes les valeurs sont considérées comme chaîne de caractères [une conversion automatique des chiffres est possible, mais elle ne fonctionne pas si le point décimal est « , » - mieux vaut une conversion explicite avec float() ]

## Lecture du format CSV – Structure de dictionnaire

```
#ouverture en lecture
f = open("personnes.csv", "r")

#importation du module csv    import csv

#lecture
lecteur = csv.DictReader(f, delimiter=";")

#affichage
for ligne in lecteur:
    print(ligne)

#fermeture
c. close()
```

Chaque ligne est  
un Dict

```
{'poids': '65', 'age': '45', 'taille': '178'}
{'poids': '89', 'age': '58', 'taille': '176'}
{'poids': '85', 'age': '35', 'taille': '194'}
```

### Remarques :

1. La première ligne est reconnue comme nom de champs
2. On utilise les clés pour accéder aux valeurs. Pour un accès indicé, une solution possible serait de convertir la collection des valeurs [ ligne.values() ] en liste.

# **FORMAT JSON**

Fichier structuré

## Format JSON

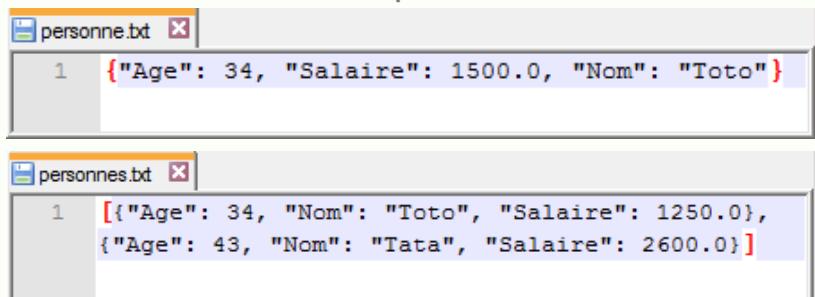
Un fichier JSON (JavaScript Object Notation) est un fichier texte (!) mais avec une structure standardisée permettant de rendre compte de l'organisation des données. C'est un format reconnu pour l'échange de données entre applications. Deux types d'éléments structurels : (1) des ensembles de paires « nom – valeur » ; (2) des listes ordonnées de valeur.

```
#début définition
class Personne:
    """Classe Personne"""
#constructeur
def __init__(self):
    #lister les champs
    self.nom = ""
    self.age = 0
    self.salaire = 0.0
#fin constructeur
... saisie et affichage ...
#fin définition
```

### A faire :

- (A) Comment sauvegarder un objet de type Personne dans un fichier ?
- (B) Comment sauvegarder une collection (liste) de personnes.

### Exemples



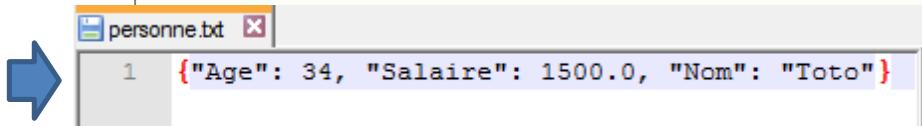
The screenshot shows two windows of a text editor. The top window is titled 'personne.txt' and contains a single line of JSON: `1 {"Age": 34, "Salaire": 1500.0, "Nom": "Toto"}`. The bottom window is titled 'personnes.txt' and contains two lines of JSON: `1 [{"Age": 34, "Nom": "Toto", "Salaire": 1250.0}, {"Age": 43, "Nom": "Tata", "Salaire": 2600.0}]`.

## Le plus simple : passer par le type dictionnaire

Idée : Le type dict permet de définir une collection d'éléments prenant la forme « étiquette : valeur ». Nous exploitons cela pour sauver les champs de l'objet.

```
# -*- coding: utf -*-
import Personne
import ModulePersonne as MP
import module json
import json
#saisie personne
p = MP.Personne()
p.saisie()
#sauvegarde
f = open("personne.json","w")
#dictionnaire
d = {"Nom":p.nom,"Age":p.age,"Salaire":p.salaire}
#sauver au format json
json.dump(d,f)
#fermer le fichier
f.close();
```

- noter l'importation du module json
- une étape clé est la transformation de l'objet référencé par p en un dictionnaire référencé par d
- la fonction dump() de la classe json permet de stocker l'objet dictionnaire dans le fichier référencé par f
- il ne faut pas oublier d'ouvrir en écriture puis de fermer le fichier
- noter le format de fichier json avec les accolades {} pour délimiter un enregistrement



Remarque : Passer par un dictionnaire est un artifice destiné à nous faciliter la vie. De manière plus académique, il faudrait rendre l'objet directement sérialisable c.-à-d. quand on fait dump() dessus, les informations sont correctement inscrites dans le fichier. A voir en M2 mon cours de C#.

## Sauvegarde : astuce, utiliser l'attribut standard `__dict__`

```
# -*- coding: utf -*-
import Personne
import ModulePersonne as MP
#import module json
import json
#saisie personne
p = MP.Personne()
p.saisie()
#sauvegarde
f = open("personne.json","w")
#sauver au format json
json.dump(p.__dict__,f) #!!!
#fermer le fichier
f.close();
```

Idée : Tous les objets Python (instance de classe) possède l'attribut standard `__dict__`, il recense les champs de la classe et leurs valeurs pour l'objet. Il est de type dictionnaire. C'est exactement ce qu'il nous faut.

← le code est grandement simplifié

## Charger l'objet via le type dictionnaire

```
# -*- coding: utf -*-
# import Personne
import ModulePersonne as MP
#import module json
import json
#ouverture fichier
f = open("personne.json","r")
#chargement
d = json.load(f)
print(d)
print(type(d))
#transf. en Personne
p = MP.Personne()
p.nom = d["Nom"]
p.age = d["Age"]
p.salaire= d["Salaire"]
#affichage
p.affichage()
#fermeture
f.close();
```

- le fichier est ouvert en lecture maintenant
- `load()` de la classe `json` s'occupe du chargement
- l'objet chargé est de type `dict` (dictionnaire)
- nous créons une instance de `Personne`, et nous recopions les informations
- l'objet référencé par `p` peut vivre sa vie par la suite Ex. ici utilisation de la méthode `affichage()`

```
{'Salaire': 1500.0, 'Nom': 'Toto', 'Age': 34}
<class 'dict'>
Son nom est Toto
Son âge : 34
Son salaire : 1500.0
```

```

import Personne
import ModulePersonne as MP
import module json
import json
#liste vide
liste = []
#nb. de pers ?
n = int(input("Nb de pers :"))
    #saisie liste
for i in range(0,n):
    a = MP.Personne()
    a.saisie()
    liste.append(a)
#sauvegarde
f =
open("personnes.json","w")
tmp = [] #une liste temporaire
#pour chaque personne
for p in liste:
    #créer un dictionnaire
    d = {}
    d["Nom"] = p.nom
    d["Age"] = p.age
    d["Salaire"] = p.salaire
    #ajouter dans liste tmp
    tmp.append(d)
#sauvegarde de la liste tmp
json.dump(tmp,f)
#fermer le fichier
f.close();

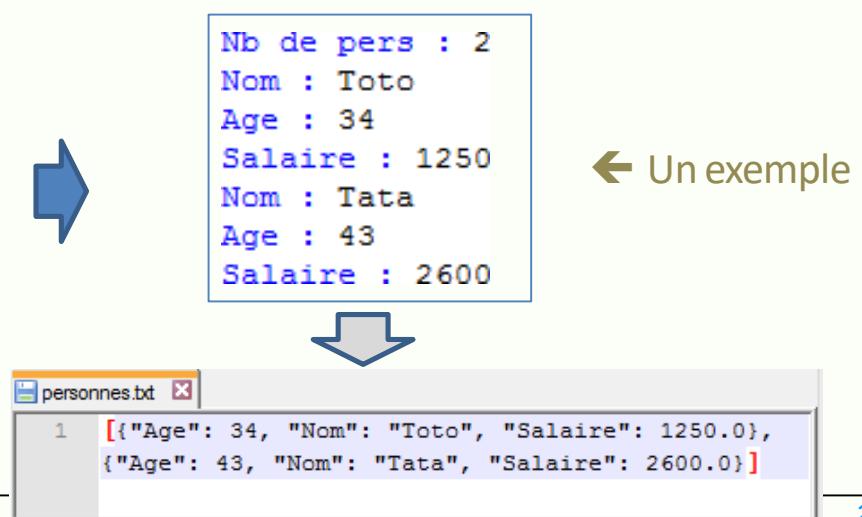
```

## Sauver une liste d'objets

Objectif: Sauvegarder un ensemble de personnes dans un fichier JSON.

Idée: Utiliser une double collection. Une liste (list) dans laquelle sont insérés des dictionnaires (dict).

- format JSON : noter l'imbrication des [ ] et { } pour délimiter la liste et chaque enregistrement
- effectuer un `dump()` sur la liste principale `tmp` revient à sauver chaque dictionnaire (Personne) qui le compose.



```

import Personne
import ModulePersonne as MP

import module json
import json

#ouverture fichier
f = open("personnes.json","r")

#chargement
tmp = json.load(f)

#conv. en liste de personnes
liste = []
for d in tmp:
    #créer une personne
    p = MP.Personne()
    p.nom = d["Nom"]
    p.age = d["Age"]
    p.salaire= d["Salaire"]
    #affichage
    p.affichage()
    #l'ajouter dans la liste
    liste.append(p)

print("Nb personnes : ",len(liste)) #fermeture
f.close();

```

## Charger une liste d'objets

Objectif: Charger un ensemble de personnes à partir d'un fichier JSON.

Idée: Convertir les dict en objet de type Personne

Effectuer un `load()` permet de charger la liste de dictionnaires, référencée par `tmp`. Une instance de Personne est créée pour chaque élément de type dict, les informations sont recopiées.



```

Son nom est Toto
Son âge : 34
Son salaire : 1250.0
Son nom est Tata
Son âge : 43
Son salaire : 2600.0
Nb personnes : 2

```

