6

Structures algorithmiques

6.1 Compétences exigées

— Objectifs —

Les principales capacités développées dans cette partie sont les suivantes :

- concevoir l'en-tête (ou la spécification) d'une fonction, puis la fonction elle-même,
- traduire un algorithme dans un langage de programmation,
- gérer efficacement un ensemble de fichiers correspondant à des versions successives d'un fichier source,
- rechercher une information au sein d'une documentation en ligne, analyser des exemples fournis dans cette documentation,
- documenter une fonction, un programme plus complexe.

6.2 Généralités

L'**indentation du code est fondamentale en *Python*** puisqu'elle détermine la structure du code.

Comme déjà signalé auparavant, les mots-clefs et les identifiants sont tous sensibles à la casse des caractères : `foo`, `Foo`, `fOO` et `FOO` sont des variables différentes.

Les caractères suivant un # sont des commentaires, destinés aux autres lecteurs du code source..

- Il est possible d'appeler l'interpréteur *Python* à partir d'un environnement de développement intégré. Sous *spyder*, *geany* et *idle* cela se fait par F5, sous *Eclipse* par F11 ou CTRL - F11. Il est possible de préciser les options de ligne de commande.
  On peut préciser les paramètres à passer au script dans le menu d'exécution.
- Pour exécuter un script *exemple.py* à partir d'une terminal on tape simplement : `python exemple.py`
  Sur un système *Unix/Linux*, si la première ligne du script est de la forme `#!/usr/bin/python` ou `#!/usr/bin/python3` et que l'utilisateur a des droits en exécution dessus, il suffit de taper `exemple.py`.
  La deuxième ligne (ou la première si la précédente est absente) `# -*- coding: utf-8 -*-` est particulière : elle indique à la machine virtuelle *Python* que le code source est écrit en *UTF-8*, ce qui permet les caractères accentués.

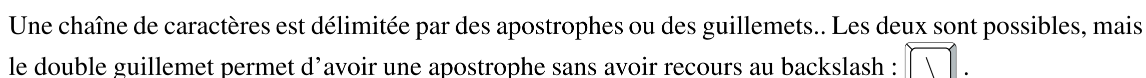
Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

6.3 Types de données dynamiques

6.3.1 Chaînes de caractères

Le type chaîne de caractères est nommé `str` en *Python* (`str` comme *string*).
Une chaîne est en fait une chaîne de caractères qui est utilisée pour stocker et représenter de l'information textuelle. D'un point de vue fonctionnel, les chaînes peuvent représenter tout ce qui peut être encodé comme du texte.
Un caractère en *Python* est une chaîne d'un caractère. **Les chaînes de caractères sont des séquences non modifiables** : elle répondent aux opérations habituelles mais elle ne peuvent pas être modifiées, elles sont **non mutables**.

Une chaîne de caractères est délimitée par des apostrophes ou des guillemets.. Les deux sont possibles, mais le double guillemet permet d'avoir une apostrophe sans avoir recours au backslash : \ .



Le triple double guillemet permet d'entrer une chaîne de caractères sur plusieurs lignes, y compris les caractères de retour de ligne.

On peut effectuer quelques opérations à partir des chaînes, par exemple :

Chaîne	Interprétation
`ch1="`	chaîne vide
`ch2=' '`	chaîne vide
`ch3= "bla"`	double guillemets
`bloc= """...."""`	bloc

À partir de ces chaînes, on peut effectuer par exemple :

```
>>> ch2=' '
>>> ch3="bla"
>>> ch3+ch2+ch3
'bla bla'
>>> ch3*5
'blablablablabla'
```

Illustration du caractère non modifiable ou non-mutable des chaînes :

```
>>> ch='Rello !'
>>> ch[0]='H'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

mais

```
>>> ch='H'+ch[1:]
>>> ch
'Hello !'
```

L'ensemble des méthodes pour les chaînes est rassemblée en annexe page 306. Certaines sont utilisées dans l'exemple ci-dessous :

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

```
>>> x = "physique"
>>> x.upper()
'PHYSIQUE'
>>> x
'physique'
>>> x[0].upper()+x[1:]
'Physique'
>>> x
'physique'
>>> x.capitalize()
'Physique'
>>> matieres="physique chimie informatique"
>>> matieres.capitalize()
'Physique chimie informatique'
>>> matieres.title()
'Physique Chimie Informatique'
>>> matieres.swapcase()
'PHYSIQUE CHIMIE INFORMATIQUE'
>>> matieres.islower()
True
>>> matieres.isupper()
False
>>> matieres.isalpha()
False
>>> matieres.isdigit()
False
>>> matieres.split()
['physique', 'chimie', 'informatique']
>>> matieres.find("for")
18
>>> matieres.find("phy")
0
>>> matieres.find("hys")
1
>>> matieres.replace("i","o")
'physoque chomoe onformatoque'
>>> matieres.count("ique")
2
>>> for mot in matieres.split():
    print(mot)
physique
chimie
informatique
```

On peut découper une chaîne de caractères en mots par la méthode `split()`.

```
>>> chaine="La méthode split est très pratique"
>>> chaine.split()
['La', 'méthode', 'split', 'est', 'très', 'pratique']
```

On peut spécifier un délimiteur à `split` :

```
>>> chaine="La méthode split est très pratique"
>>> chaine.split('r')
['La méthode split est t', 'ès p', 'atique']
```

Il est possible de référencer l'un des caractères en considérant la chaîne comme une liste (`chaine[3]` pour le 4e caractère), mais chaque caractère est lui-même de type *str*.

```
>>> chaine="La méthode split est très pratique"
>>> chaine
'La méthode split est très pratique'
>>> chaine[3]
'm'
>>> print(type(chaine[3]))
<class 'str'>
```

Des outils plus évolués de manipulation de chaînes de caractères sont accessibles par le module `re` (cf. paragaraphe 16.2 page 230).

6.3.2 Listes

— Liste —

Une liste permet de stocker des successions de valeurs, et de les retrouver par leur index dans sa structure. Elle peut être modifiée : modification des éléments présents, suppression, et ajout d'élément(s).

Une `liste` s'écrit entre crochets : `tab = [6, 7]`.

La numérotation des cases commence à 0. L'index se place entre crochets : `tab[4]`. Il est possible de sélectionner des tranches de ces structures : `tab[1:4]`, `tab[:5]`, `tab[3:]`, `tab[1:-1]`. Le premier élément est inclus, le dernier est exclu : `tab[1:3]` contient les cases 1 et 2. L'élément d'indice `-1` est le dernier élément de la liste, l'élément d'indice `-2` le pénultième, . . .

Il n'est pas possible de créer une nouvelle case à une liste par une simple affectation. Si `tab` est de taille 6, `tab[8]=42` entraîne une erreur. Il faut ajouter la nouvelle valeur par `tab.append(42)`. On peut concaténer `liste2` à `liste1` par `liste1.extend(liste2)`.

L'opérateur `len()` donne le nombre d'éléments de la liste : `len([3, 7, [5, 3], "Last"])` renvoie 4.

On peut supprimer un élément d'une liste en connaissant son index `i` par `del tab[i]`. On peut supprimer le premier élément ayant une certaine valeur `val` par `tab.remove(val)`.

`tab.remove(tab[i])` ne supprime donc pas forcément l'élément d'index `i`.

La méthode `pop()` supprime et renvoie le dernier élément d'une liste :

```
>>> liste=[9,6,42]
>>> x=liste.pop()
>>> liste
[9, 6]
>>> x
42
```

La variable `x` vaut désormais 42, et `liste` vaut [9, 6].

La méthode `insert(i, x)` insère l'élément `x` à l'indice `i`. Si `i` est supérieur ou égal à la taille de la liste, `x` est inséré en dernière position, comme un appel à `append(x)`.

```
>>> liste = [9, 6, 42]
>>> liste.insert(1, 7)
>>> liste.insert(100, 24)
>>> liste
[9, 7, 6, 42, 24]
```

La variable `liste` vaut désormais `[9, 7, 6, 42, 24]`.

On peut tester l'appartenance d'un élément à une liste par l'opérateur `in` .

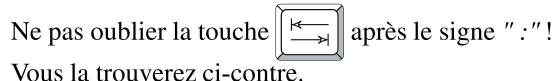
— Remarque —

Il est possible d'inverser le comportement de `in` en le remplaçant par `not in`.

```
>>> premiers = [2, 3, 5, 7, 11, 13, 17, 19]
>>> if 5 in premiers:
...     print ("%i est un nombre premier" %5)

5 est un nombre premier
>>> if 4 not in premiers:
...     print("%i n'est pas un nombre premier" %4)

4 n'est pas un nombre premier
```



Ne pas oublier la touche [tab] après le signe ":"!
Vous la trouverez ci-contre.

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

Un tableau à n dimensions n'est rien d'autre qu'une liste de listes qui sont toutes de la même longueur.

Pour faire une liste de n listes vides, il ne faut pas écrire `tab = n*[[]]`, car on aurait n copies de la même liste :

```
>>> tab = 4*[[]]
>>> tab[1].append(42)
>>> tab
[[42], [42], [42], [42]]
```

mais :

```
>>> tab=[]
>>> for i in range(4):
...     tab.append([])
...
>>> tab[1].append(42)
>>> tab
[[], [42], [], []]
```

Une variable de type `list` référence une zone mémoire. Si deux variables référencent une même zone mémoire, modifier l'une impacte l'autre :

```
>>> liste1 = ["Terre", 245, "Lune"]
>>> liste2 = liste1
>>> liste1.insert(2,"Vénus")
>>> liste1
['Terre', 245, 'Vénus', 'Lune']
>>> liste2
['Terre', 245, 'Vénus', 'Lune']
```

Extraire une tranche d'une liste en réalise une copie d'une partie de son contenu (éventuellement la liste entière). Ce comportement est différent des tableaux de *numpy* (cf. paragraphe 18.4.3 page 249).

```
>>> liste1 = ["Terre", 245, "Lune"]
>>> liste2 = liste1[:]
>>> liste1.insert(2,"Vénus")
>>> liste1
['Terre', 245, 'Vénus', 'Lune']
>>> liste2
['Terre', 245, 'Lune']
```

On peut aussi effectuer une copie de liste avec la fonction `list` :

```
>>> liste1 = ["Terre", 245, "Lune"]
>>> liste2 = list(liste1)
>>> liste1.insert(2,"Vénus")
>>> liste1
['Terre', 245, 'Vénus', 'Lune']
>>> liste2
['Terre', 245, 'Lune']
```

L'ensemble des méthodes pour les listes est rassemblé en annexe page 307.

⇒ Activité 6. 18

Indiquez les résultats donnés par l'exécution des codes suivants :

1 -

```
>>> list1=["maths","physique","chimie"]
>>> list1.sort()
>>> list1.reverse()
>>> list1
```

2 -

```
>>> list2=["maths","français","anglais","méca","élec","physique"]
>>> list2.pop(3)
>>> list2
>>> len(list2)
>>> list2[0:4:2]
```

3 -

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

```
>>> list3=list(range(2,19,2))
>>> list4=list3
>>> del(list4[3])
>>> list3.reverse()
>>> list4.remove(10)
>>> list3
>>> list4
```

4 -

```
>>> phrase = ['Python ','c','"'",'est ','du ','béton !']
>>> for mot in phrase:
    print(mot, end ='')
```

Il est possible de récupérer une liste grâce à la suite d'instructions `list(eval())`. Les éléments doivent être séparés par des virgules :

```
>>> ch = input("Votre réponse : ")
Votre réponse : 3,5,6
>>> list(eval(ch))
[3, 5, 6]
```

6.3.3 Tuples

— Tuple —

Un tuple permet de stocker des successions de valeurs, et de les retrouver par leur index dans la structure, mais ne peut pas être modifié : on dit qu'il est *non-mutable*.

Un `tuple` s'écrit entre parenthèses : `tup = (6, 7)`. Il peut contenir tout type de données non-mutable, y compris d'autres tuples : `tup = (6, 7.05, "poisson", (-5, "dauphin"))`.

On accède aux éléments d'un tuple par leurs index ou leurs tranches d'index exactement comme pour une liste. La fonction `len()` et l'opérateur `in` s'appliquent également.

Il n'est pas forcément nécessaire d'utiliser les tuples mais comme certaines méthodes ou fonctions renvoient des tuples, il faut connaître leurs propriétés.

L'ensemble des méthodes pour les tuples est rassemblé en annexe page 308.

6.3.4 Dictionnaires

Le dictionnaire est un système très souple pour intégrer des données. Si on pense aux listes comme une collection d'objets ordonnés et portant un indice par rapport à la position, un dictionnaire est une liste comportant à la place de ces indices, un mot servant de clé pour retrouver l'objet voulu.
Un dictionnaire est affecté par une paire d'accolades ({ }), par exemple :

```
>>> capitales = {'France' : 'Paris', 'Espagne' : 'Madrid'}
```

Les clés peuvent être de toutes les sortes d'objet non modifiables. Par contre la valeur stockée peut être de n'importe quel type.
Un dictionnaire n'est pas ordonné comme une liste, mais c'est une représentation plus symbolique de classement par clés. La taille d'un dictionnaire peut varier, et le dictionnaire est un type modifiable ou mutable.

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

Opération	Interprétation
`d1 = {}`	Dictionnaire vide
`d2={'one' : 1, 'two' : 2}`	Dictionnaire à deux éléments
`d3= {'count ': {'one': 1, 'two': 2}}`	Inclusion
`d2 ['one'], d3['count']['one']`	Indiçage par clé
`d2.has_keys('one')`	Méthode : test d'appartenance
`d2.keys()`	liste des clés
`d2.values()`	Liste des valeurs
`len(d1)`	Longueur (nombre d'entrées)
`d2[cle] = [nouveau]`	ajout ou modification
`del d2[cle]`	destruction

TABLE 6.1 – Dictionnaires

Exemple plus complet :

```
>>> tel = { 'Pierre' : 9876, 'Paul': 8765, 'Jacques' : 7654 }
>>> print(tel)
{'Paul': 8765, 'Pierre': 9876, 'Jacques': 7654}
>>> print(tel['Paul'])
8765
>>> tel['Henri']=6543
>>> print(tel)
{'Paul': 8765, 'Pierre': 9876, 'Henri': 6543, 'Jacques': 7654}
>>> print(tel.keys())
dict_keys(['Paul', 'Pierre', 'Henri', 'Jacques'])
>>> print(tel.values())
dict_values([8765, 9876, 6543, 7654])
>>> print(tel.items())
dict_items([('Paul', 8765), ('Pierre', 9876), ('Henri', 6543), ('Jacques', 7654)])
>>> print(len(tel))
4
>>> del tel['Paul']
>>> print(len(tel))
3
```

6.3.5 Les ensembles

Les ensembles en python se définissent grâce à la commande *set*. On distingue 2 types d'ensembles :

- les `set` : modifiables,
- les `frozenset` : non modifiables.

Certaines opérations sont très pratiques.

Définissons tout d'abord 3 ensembles :

```
>>> ens1=set([1,2,3,4,5,6,7,8,9])
>>> ens2=set([2,3,4,5])
>>> ens3=set([5,6])
```

6.3.5.1 Inclusion

L'inclusion, notée en maths $\subset$ peut se faire avec la méthode `issubset` :

```
>>> ens2.issubset(ens1)
True
```


Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

6.3.5.2 Réunion

La réunion, notée en maths $\cup$ peut se faire avec la méthode `union()`, ou avec l'opérateur | :

```
>>> ens2.union(ens3)
{2, 3, 4, 5, 6}
>>> ens2 | ens3
{2, 3, 4, 5, 6}
```

6.3.5.3 Intersection

L'intersection, notée en maths $\cap$ peut se faire avec la méthode `intersection`, ou avec l'opérateur & :

```
>>> ens1 & ens2
{2, 3, 4, 5}
>>> ens1 & ens3
{5, 6}
>>> ens2 & ens3
{5}
>>> ens2.intersection(ens3)
{5}
```

6.3.5.4 Différence

La différence, notée en maths $\setminus$ peut se faire avec la méthode `difference()`, ou avec l'opérateur $-$:

```
>>> ens1 - ens2
{8, 1, 9, 6, 7}
>>> ens1.difference(ens2)
{8, 1, 9, 6, 7}
```

6.3.5.5 Différence symétrique

La différence symétrique, notée en maths Δ peut se faire avec la méthode *symmetric_difference()*, ou avec l'opérateur ^. Cette différence symétrique correspond à l'union moins les éléments communs.

```
>>> ens2^ens3
{2, 3, 4, 6}
>>> ens2.symmetric_difference(ens3)
{2, 3, 4, 6}
```

6.4 Affichage

6.4.1 La fonction *input*

Pour réaliser une saisie à l'écran, la fonction *input()* est tout indiquée : elle interrompt le programme, affiche une éventuelle invite et attend que l'utilisateur entre une donnée et la valide par *Entrée* : ENTER .

La fonction `input()` effectue toujours une saisie en mode texte (la saisie est alors une chaîne) dont on peut ensuite changer le type (on dit aussi transtyper).

```
>>> age=input("Entrez votre age : ")
Entrez votre age : 25
>>> print(type(age))
<class 'str'>
```

Il faut alors transtyper la variable *age* :

```
>>> age=int(age)
>>> print(type(age))
<class 'int'>
```

ou plus directement :

```
>>> age=int(input("Entrez votre age : "))
Entrez votre age : 25
>>> print(type(age))
<class 'int'>
```


Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

Informatique CPGE TSI - Etablissement Saint Joseph - LaSalle

6.4.2 La fonction *print*

La fonction `print()` permet l'écriture sur la sortie standard. Lors d'un appel à `print(x)`, si x n'est pas une chaîne de caractères, `x` est converti en chaîne de caractères par `str(x)`.

Il est possible de demander l'affichage de plusieurs objets en les séparant par des virgules :

```
>>> print(6, "x", 7, " = ",6*7)
6 x 7  =  42
```

Si le dernier objet est suivi de *end=”*, il n'y a pas de retour à la ligne (Attention, le retour à la ligne est évité en mettant une virgule à la fin).

On peut choisir de mettre un espace *end=' '* ou tout autre caractère.

```
>>> print(6, "x", 7, " = ",6*7, end='------------\n')
6 x 7  =  42------------
```

Certains caractères spéciaux, comme les tabulations ou les retours à la ligne, sont codés par caractères échappés : `'\t'` pour une tabulation, `'\n'` pour un retour à la ligne. Il faut échapper l'anti-slash par `'\\'` pour l'inclure dans une chaîne de caractères.

Ce caractère antislash \ permet de donner une signification spéciale à certaines séquences :

Séquence	Signification
\	saut de ligne ignoré
\	antislash
\'	apostrophe
\"	guillemets
\a	sonnerie (bip)
\b	retour arrière
\f	saut de page
\n	saut de ligne
\r	retour début de ligne
\t	tabulation horizontale
\v	tabulation verticale
\N{nom}	caractère sous forme Unicode avec nom
\uhhhh	caractère sous forme Unicode 16 bits
\uhhhhhhhh	caractère sous forme Unicode 32 bits
\ooo	caractère sous forme de code octal
\xhh	caractère sous forme de code hexadécimal

TABLE 6.2 – Séquences d'échappement

On pourra par exemple étudier les instructions et résultats suivants :

```
>>> table = " Nom\t| Age\nMar\\/in\t|60E9"
>>> print (table)
 Nom     | Age
Mar\/in |60E9
```

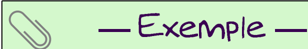
On peut demander à *Python* de ne pas interpréter les échappements dans une chaîne en la précédant par `r`.

```
>>> table = r" Nom\t| Age\nMar\\/in\t|60E9"
>>> print (table)
 Nom\t| Age\nMar\\/in\t|60E9
```

6.4.3 Opérateur %

— Opérateur % —

L'opérateur % permet de placer une ou plusieurs valeurs dans une chaîne.




```
>>> masse=65
>>> qte=3
>>> masse_mol=masse/qte
>>> print("Masse=  %i, quantité de matière= %i,\
 masse molaire = %.3f" %(masse,qte,masse_mol))
Masse=  65, quantité de matière= 3, masse molaire = 21.667
```

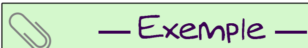
Il doit y avoir autant d'éléments dans le tuple de valeurs à insérer que de caractères % dans la chaîne de caractères. La lettre qui suit le caractère % dans la chaîne détermine le type d'information à écrire : `i` pour un nombre entier ou `d` pour le nombre de chiffres avant la virgule d'un nombre entier ou réel ; `x` pour un nombre entier hexadécimal ; `f`, `e`, `g`, `E` ou `G` pour un nombre à virgule flottante ; `s` pour une chaîne de caractères, ...
Les nombres qui précèdent permettent de maîtriser le nombre de caractères à écrire, et notamment la précision d'une nombre à virgule flottante.

Un exemple assez utile en sciences :

```
>>> a = 12369872.2587593
>>> print("%2.3e" %a)
1.237e+07
```

6.4.4 Le formatage

Cette fonction permet d'imprimer les données avec un nombre spécifique de décimales, à la manière de l'opérateur %.




```
>>> masse=2.45
>>> volume=5.9
>>> m_v=masse/volume
>>> print("Masse volumique=",m_v)
Masse volumique= 0.4152542372881356
>>> print("La masse volumique vaut {:.3f} kg/m^3".format(m_v))
La masse volumique vaut 0.415 kg/m^3
>>> print("La masse vaut : {},\nLe volume vaut : {:.2f},\n {}\
\n\tLa masse volumique vaut : {:.4f}".format(masse,volume,"-"*50,m_v))
La masse vaut : 2.45,
Le volume vaut : 5.90,
--------------------------------------------------
    La masse volumique vaut : 0.4153
```

La dernière notation permet d'arrondir les valeurs au nombre de décimales voulu, d'insérer des caractères (par exemple, des tirets ici).

Les instructions "\n" et "\t" permettent respectivement de passer à la ligne et d'insérer une tabulation horizontale.

6.4.5 Résumé

En résumé, on pourra retenir les façons suivantes, illustrées dans le programme *formatage.py* pour afficher des données :

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

```
1 # -*- coding: utf-8 -*-
2
3 print("Vous avez %i euros" % 15)
4 a = 20
5 print("Vous avez %i euros" % a)
6 print("Vous avez %i %s" % (a, "euros"))
7 print("Vous avez {} euros".format(a))
8 print("Vous avez {0} euros".format(a))
9 print("Vous avez {1} {0}".format("euros", a))
10 print("Vous avez %.2f euros" %a)
11 print("Vous avez %1.2e euros" %a)
```

Ce programme donne :

```
Vous avez 15 euros
Vous avez 20 euros
Vous avez 20 euros
Vous avez 20 euros
Vous avez 20 euros
Vous avez 20 euros
Vous avez 20.00 euros
Vous avez 2.00e+01 euros
```

6.5 Boucles et conditionnelles

6.5.1 Conditionnelle

Le branchement s'écrit :

```
if condition 1:
    instructions 1
elif condition 2:
    instructions 2
else:
    instructions 3
```

Programme *condition.py* :

```
1 # -*- coding: utf-8 -*-
2
3 x = 42
4 if x != int(x):
5     print ("Non entier")
6 elif x<0:
7     print ("Entier négatif")
8 else:
9     print ("Entier positif")
```

Entre `if` et les deux-points, il est possible de mettre toute expression booléenne, issue par exemple d'un opérateur de comparaison comme ==, <, >, <=, >= ou !=. Notez que l'opérateur de test d'égalité est == alors que l'opérateur d'affectation est =. Le test de non-égalité s'écrit != . L'opérateur de test d'appartenance `in` construit également une expression booléenne.

Il est possible de combiner des expressions booléennes par `and`, `or` et `not`.

On peut aussi écrire une condition de façon plus compacte si le code est simple :

Programme *condition-compacte.py* :

```
1 # -*- coding: utf-8 -*-
2
3 print("Entrez 2 entiers :")
4 x=int(input("x ? "))
5 y=int(input("y ? "))
6
7 # méthode classique :
```


Informatique CPGE TSI - Efbriassemant Saint Joseph - LaSalle

```
8  if x < y:
9      petit = x
10 else:
11     petit = y
12
13 # méthode compacte :
14 petit = x if x < y else y
15
16 print("\nLe plus petit est", petit)
```

— pass —

L'instruction `pass` ne fait rien, et permet éventuellement de remplir une obligation syntaxique (*Python* n'accepte pas les blocs vides).

```
if x<0 or x>=20:
    pass
else:
    print ("%i possède une valeur comprise dans l'intervalle\
    [0, 20[" %x)
```

Notez que prendre le complémentaire de l'expression booléenne fonctionne aussi, et est plus élégant :

— complémentaire —

```
if not(x<0 or x>=20):
    print ("%i possède une valeur comprise dans l'intervalle\
    [0, 20[" %x)
```

ou encore :

```
if x>=0 and x<20:
    print ("%i possède une valeur comprise dans l'intervalle\
    [0, 20[" %x)
```

Il n'existe pas d'équivalent du `switch ... case` courant dans d'autres langages. Il convient d'utiliser une succession de `elif` pour choisir entre de multiples actions possibles en fonction de la valeur d'une variable.

⇒ **Activité 6.19**

Écrivez un programme qui demande des notes (par exemple Maths, Informatique, Physique et Chimie (le reste étant quantité négligeable ☺) puis calcule la moyenne de ces notes et affiche alors "Menteur", "Félicitations", "Mention Très Bien", "Bien", "Assez Bien", "Passable", "Rattrapage", "Recalé" suivant que la moyenne est supérieure à 20, comprise entre 18 et 20, comprise entre 16 et 18, comprise entre 14 et 16, entre 12 et 14, entre 10 et 12, entre 8 et 10, inférieure à 8.

Programme *mention.py* :

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

6.5.2 Type booléen

Il est également possible d'affecter la valeur d'une expression booléenne à une variable, lequel est donc de type `bool`, et peut prendre uniquement les deux valeurs `True` et `False`.

```
>>> a=2
>>> b=3
>>> print(a==b)
False
>>> print(True or False)
True
>>> print(True and False)
False
>>> maliste=["As", "Valet", 10]
>>> "As" in maliste
True
```

6.5.3 Boucles *for* et *while*

6.5.3.1 Boucle *for*

— Boucle for —

La boucle `for` s'écrit :

```
for var in table:
    instructions
```

Le corps de la boucle est exécuté une fois pour chaque élément de `tab`, lequel peut être tout objet contenant une succession d'éléments. Il peut donc s'agir d'une liste, mais aussi d'une table de hachage, d'un tuple, d'un fichier, ou d'une fonction itératrice comme `range()`.

La fonction `range()` permet (entre autres) de générer une succession d'indices pour `for`, mais aussi toute succession de nombres : `range(début, fin, pas)`.

Ainsi, l'exemple suivant comporte deux boucles. x prend successivement les valeurs contenues dans les cases de tab. L'indice i prend successivement toutes les valeurs entières de 0 à len(tab)-1 puis len(tab)-3, c'est-à-dire les valeurs d'index valides pour tab.

Programme *boucles.py* :

```
1  # -*- coding: utf-8 -*-
2
3  tab = [6, 7, 2, 8, 42, 14]
4
5  for i in range(len(tab)):
6      print (i, tab[i])
7
8  print()
9
10 for i in range(len(tab)-2):
11     print (i, tab[i])
```

⇒ **Activité 6. 20**

Affichez les entiers de 0 à 31 (31 non compris), de trois en trois, en utilisant une boucle for et l'instruction range().

Programme *for-range.py* :

⇒ **Activité 6. 21**

Affichez chaque caractère de la chaîne "bonjour" en utilisant une boucle for. Affichez chaque élément de la liste [”*hello*”, [2,6], 3.1416] en utilisant une boucle for.

Programme *boucle-for.py* :

6.5.3.2 Boucle *while*

— Boucle while —

La boucle while s'écrit :

```
while cond:
    instructions
```


Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

Le corps de la boucle est exécuté chaque fois que la condition est vraie, et la boucle s'arrête dès que la condition est testée comme fausse. Il est donc possible que le corps de la boucle ne soit jamais exécuté.

⇒ **Activité 6. 22**
À l'aide d'une boucle `while`, calculez la somme d'une suite de nombres non nuls entrés par l'utilisateur.
Comptez combien il y a de données et combien sont supérieures à 100.
Un nombre égal à 0 indique la fin de la boucle.
Programme *somme100.py* :

⇒ **Activité 6. 23**
Écrivez, à l'aide d'une boucle `while not`, un programme qui demande à l'utilisateur d'entrer un entier pair et qui ne s'arrête que lorsque c'est le cas.
Programme *while-not.py* :

6.5.4 Boucle définissant une liste

Il est immédiat de faire une boucle qui remplit une liste par des appels successifs à `append()` :

```
>>> y = []
>>> for x in range(-10, 10, 1):
    y.append(x**3)
```

Toutefois ces appels à `append()` sont assez coûteux en temps de calcul, et la syntaxe est assez lourde. *Python* propose une alternative plaçant `for` à l'intérieur de la définition de la liste (cf. page 73).

```
>>> y = [x**3 for x in range(-10, 10, 1)]
```

6.5.5 break et continue

— Break et continue —

Dans les boucles, l'instruction `break` sort de la plus petite boucle `for` ou `while` englobante.
L'instruction `continue`, continue sur la prochaine itération de la boucle.

Informatique CPGE TSI - - Établissement Saint Joseph - LaSalle

Exemples :

Programme *break_continue.py* :

```
1  # -*- coding: utf-8 -*-
2
3  tab = [6, 7, 2, 8, 42, 14, 12, 21]
4  for x in tab:
5      if 42%x!=0:
6          continue
7      print (x, end=' ')
8  print()
9
10 i = 0
11 while True: # ou while 1:
12     if i==len(tab)-4:
13         break
14     print ("position",i,"-> élément",tab[i])
15     i += 1
```

Les instructions de boucle ont une clause `else` ; elle est exécutée lorsque la boucle se termine par épuisement de la liste (avec `for`) ou quand la condition devient fausse (avec `while`), mais pas quand la boucle est interrompue par une instruction `break`. Ceci est expliqué dans la boucle suivante, qui recherche des nombres premiers :

Programme *break-e.py* :

```
1  # -*- coding: utf-8 -*-
2
3  for n in range(2, 10):
4      for x in range(2, n):
5          if n % x == 0:
6              print (n, "égale", x, "* {}".format(int(n/x)))
7              break
8      else:
9              print (n, "est un nombre premier")
```

qui donne :

```
2 est un nombre premier
3 est un nombre premier
4 égale 2 * 2
5 est un nombre premier
6 égale 2 * 3
7 est un nombre premier
8 égale 2 * 4
9 égale 3 * 3
```

⇒ **Activité 6. 24**

Indiquez le résultat de la suite d'instructions suivante :

Programme *break-act.py* :

```
1  # -*- coding: utf-8 -*-
2
3  for x in range(1, 11):
4      if x == 5:
5          break
6      print(x)
7
8  print("Boucle terminée si x = ", x)
```

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

Informatique CPGE TSI - - Établissement Saint Joseph - LaSalle

⇒ **Activité 6. 25**

Indiquez le résultat de la suite d'instructions suivante :

Programme *continue-act.py* :

```
1  # -*- coding: utf-8 -*-
2
3  for x in range(1, 6):
4      if x == 2:
5          continue
6      print(x)
7
8  print("Le 2 est passé")
```

⇒ **Activité 6. 26**

Indiquez le résultat de la suite d'instructions suivante :

Programme *break.py* :

```
1  # -*- coding: utf-8 -*-
2
3  while 1: # 1 est toujours vrai -> boucle infinie
4      lettre = input("Tapez 'Q' pour quitter : ")
5      if lettre == "Q":
6          print("Fin de la boucle")
7          break
```

6.6 Fonctions et procédures

Les fonctions et procédures sont primordiales pour la programmation procédurale. Il faut :

- comprendre pourquoi on écrit des fonctions,
- s'obliger à le faire dès les premiers programmes.

La différence essentielle entre les deux, c'est que :

- Une fonction effectue un calcul et vaut quelque chose.
- À l'inverse, une procédure n'a pas de valeur mais effectue une tâche (afficher, modifier un objet par exemple).

6.6.1 Notion de fonction

Il est bien sûr possible de créer ses propres fonctions, qui reçoivent éventuellement des paramètres à traiter, et renvoient un résultat via `return`.

La fonction se déclare avec le mot-clé `def`.

Il faudra distinguer la définition de la fonction et son utilisation :

- définition : on indique comment "marche" la fonction et comment on s'en sert (déclaration). La fonction n'est pas utilisée (exécutée), mais juste "lue" pour être connue de l'interpréteur.
- utilisation : on exécute de manière effective le code de la fonction pour des valeurs d'entrées fixées.

⇒ **Activité 6. 27**

Écrivez une fonction qui calcule le montant *TTC* à partir d'un montant *HT*.

Programme *fonc-ttc.py* :

— Remarque —

L'instruction `return` termine la fonction, même si cette instruction est exécutée avant la fin du texte de la fonction.

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

Un autre exemple de fonction :

Programme *racines2.py* :

```python
# -*- coding: utf-8 -*-

def discr(a, b, c):
    return b**2-4*a*c

def racines(a, b, c):
    delta = discr(a, b, c)
    if delta > 0:
        return (-b-delta**0.5)/(2*a), (-b+delta**0.5)/(2*a)
    elif delta == 0:
        return -b/(2*a),
    else:
        return("racines complexes")

print(racines(2,3,5))
print(racines(2,7,5))
```

⇒ **Activité 6. 28**

Déterminez le rôle du code précédent.

Voici quelques occasions dans lesquelles vous devez écrire des fonctions (ou des procédures) :

- Le bloc de programme que vous écrivez ne tient pas en entier à la vue sur votre écran.
- Vous utilisez plus d'une fois des portions de code exactement identiques, ou presque identiques. Vous gagnerez en clarté et en investissement à écrire une fonction à la place de ce bloc et à l'appeler plusieurs fois.
- Vous ne savez pas exactement si la méthode de résolution employée pour telle tâche est la meilleure. Mettez la dans une fonction et utilisez cette fonction. Si vous trouvez une meilleure façon de procéder, il suffira de modifier la fonction sans toucher au reste.
- Le problème que vous traitez est difficile : découpez-le en fonctions que vous écrirez et testerez séparément les unes des autres (le bénéfice que vous aurez à pouvoir tester très facilement des portions de programme est énorme). Dans chaque fonction indépendante, vous pourrez choisir des noms de variables appropriés, qui rendront votre code plus simple à comprendre.

⇒ **Activité 6. 29**

Écrivez, à l'aide d'une boucle `for`, une fonction qui renvoie la factorielle d'un nombre quelconque.

Programme *facto.py* :

Il est parfois intéressant d'utiliser le mot réservé `None` afin d'affecter une valeur qui ne vaut rien. Un exemple ci-dessous dans le programme *none.py* :

```
1  # -*- coding: utf-8 -*-
2
3  maliste=[1,2,3]
4
5  def test(x,a):
6      if a in x:
7          return("Yes")
8      else:
9          return(None)
10
11 print(test(maliste,3))
12 print(test(maliste,4))
```

6.6.2 Notion de procédure

En *Python*, il n'y a pas de différence syntaxique entre fonction et procédure mais il ne faut pas les confondre.

Une procédure a un effet de bord : elle modifie quelque chose qui est en dehors de la procédure.
En principe, on souhaite qu'une fonction n'ait pas d'effet de bord et c'est pour cette raison qu'on évite si possible les affichages dans une fonction.

Voici la procédure qui calcule le montant TTC à partir d'un montant HT :

Programme *proc-ttc.py* :

```
1  # -*- coding: utf-8 -*-
2
3  def proc_ttc(valht,taux):
4      """
5      Affiche le montant TTC, connaissant le montant HT (valht)
6      et le taux de TVA (taux)"""
7      valttc=valht*(1+taux/100)
8      print(valttc)
9
10 proc_ttc(120,4)
```

On pourrait penser que la fonction *fonc-ttc* et la procédure *proc-ttc* sont équivalentes. Ce serait une erreur. La fonction est plus générale. On ne peut pas écrire par exemple :

```
>>> print("Double du prix ttc : ",proc-ttc(100,19.6)*2)
```

En revanche, la même ligne, en utilisant *fonc-ttc* fournirait le bon résultat. Assurez-vous d'avoir bien compris pourquoi la ligne qui précède marche avec une fonction et pas avec une procédure. C'est vraiment très important !

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

Informatique CPGE TSI - Etablissement Saint Joseph - LaSalle

⇒ **Activité 6. 30**

Écrivez une procédure qui écrit la table de multiplication d'un nombre qu'on appellera $base$, qui commence à $debut$, qui s'arrête à fin par pas de pas.

Programme *multi.py* :

6.6.3 Documentation

Il est très important de documenter les fonctions que vous allez créer, pour vous ou pour d'autres qui vont les utiliser. Pour cela, c'est très simple, il suffit de placer immédiatement sous la ligne de définition de votre fonction, une chaîne de caractères entre triple 'quote' (""" ou ''') qui sera considérée un commentaire. On parle aussi de *docstring*.

Exemple :

```
def fonction(argument1, argument2) :
        """
        Ceci est une fonction qui ne fait rien mais qui prend
        2 arguments : argument1 et argument2
        """
        pass
```

6.6.4 Variables locales et globales

Si on affecte une valeur à une variable dans la fonction ou dans la procédure, on définit une variable *locale*. Cette variable n'existe qu'à l'intérieur de la fonction, et n'est plus accessible ensuite. Si une fonction f_1 a une variable locale *x* et appelle une fonction f_2, la variable *x* n'est pas disponible dans f_2.

Les paramètres de la fonction se comportent comme des variables locales à cette fonction.

Si une variable existe déjà à l'extérieur de la méthode avec le même nom — c'est donc une variable *globale* — la variable locale masque la variable globale. Les opérations réalisées à l'intérieur de la fonction portent uniquement sur la variable locale, et la variable globale de même nom n'est pas modifiée.

⇒ **Activité 6. 31**

Interprétez le résultat des programmes suivants :

1. Programme *variable1*.

```
1  # -*- coding: utf-8 -*-
2
3  x = 2
4  def f(x):
5      x = x + 10
6      return x
7  print(f(x))
8  print(x)
```

2. Programme *variable2*.

```
1  # -*- coding: utf-8 -*-
2
3  x = 2
4  def f(y):
5      y = y + 10
6      return y
7  print(f(x))
8  print(x)
```

3. Programme *variable3*.

```
1  # -*- coding: utf-8 -*-
2
3  x = 2
4  def f(y):
5      y = y + 10
6      return x
7  print(f(x))
8  print(x)
```

4. Programme *variable4*.

```
1  # -*- coding: utf-8 -*-
2
3  x = 2
4  def f(y):
5      y = y + 10
6      return y
7  print(f(x))
8  print(y)
9  print(type(y))
```

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

Si on souhaite modifier une variable globale à l'intérieur d'une fonction, il convient de faire référence à cette variable globale par `global`.

⇒ **Activité 6.32**

Indiquez le résultat donné par l'exécution du code suivant :

Programme *variables.py* :

```
1  # -*- coding: utf-8 -*-
2
3  def f_loc(x):
4      a = x
5      print ("f_loc : \t", a)
6
7  def f_glob(x):
8      global a
9      a = x
10     print ("f_glob : \t", a)
11
12 def g(x):
13     print ("g : \t\t", a)
14
15 a = 42
16 f_loc(33)
17 g(24)
18 f_glob(65)
19 f_loc(76)
20 g(77)
```

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

6.6.5 Local, englobant, global et built-in

Les auteurs anglophones conseillent de retenir la règle "legb", acronyme qui permet de mémoriser l'ordre de résolution des noms de variable dans *Python* : *local*, *englobant*, *global* et *built-in*.

- Les variables *locales* à une fonction sont celles définies dans cette dernière ainsi que ses paramètres. **Elles n'existent donc pas en dehors de cette fonction**,
- Les variables du bloc englobant d'une fonction sont celles dont la portée contient celle-ci,
- Les variables *globales* sont celles déclarées au niveau du fichier python,
- Enfin, *built-in* regroupe ce qui est défini à l'exécution de l'interpréteur.

— Remarque —

Dans un premier temps, il est préférable de ne pas utiliser de variable globale, ce qui est une bonne chose de toutes façons.

6.6.6 Fonction retournant plusieurs valeurs

Une fonction est capable de retourner plusieurs valeurs en une seule.
Ces valeurs peuvent être récupérées dans différentes variables ou dans un tuple.

Ainsi, le programme *fonc_var_mul.py* :

```
1  #-*- coding: utf-8 -*-
2
3  def maFonction(a,b,c):
4      return (a+b,b+c,a-c)
5
6  print(maFonction(10,5,3))
7
8  # récupération dans des variables
9  aa, bb, cc = maFonction(10,5,3)
10 print( "aa vaut %i, bb vaut %i, cc vaut %i" % (aa, bb, cc) )
11
12 # récupération via un Tuple
13 resultat = maFonction(10,5,3)
14 print( "aa est égal à %i, bb est égal à %i, cc est égal à %i" % resultat )
```

retourne :

```
(15, 8, 7)
aa vaut 15, bb vaut 8, cc vaut 7
aa est égal à 15, bb est égal à 8, cc est égal à
```

Si `return` est suivi par plusieurs paramètres, ceux-ci sont placés dans un tuple. S'il n'y a pas de `return` (cas où $\Delta < 0$ dans le programme *racines*), la valeur renvoyée est la valeur spéciale [1] `None` de type `NoneType`.

Si on souhaite garantir que *racines* renvoie toujours un tuple, même vide, il faut traiter le cas $\Delta < 0$, et forcer la création d'un tuple quand $\Delta = 0$, par l'ajout d'une virgule ou la création explicite d'un tuple avec `return tuple([-b/2/a])`.

Programme *racines2-t.py* :

```
1  # -*- coding: utf-8 -*-
2
3  # Détermine le discriminant de a x^2 + b x + c = 0
4  def discr(a, b, c):
5      return b**2-4*a*c
6
```

1. Cette valeur spéciale est appelée `null` dans de nombreux autres langages.

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

```
7   """ Détermine les racines de a x^2 + b x + c = 0 """
8   """ Renvoie un tuple """
9
10  def racines(a, b, c):
11      delta = discr(a, b, c)
12      if delta > 0:
13          return (-b-delta**0.5)/2/a, (-b+delta**0.5)/2/a
14      elif delta == 0:
15          return -b/2/a,
16      else:
17          return ()
```

6.6.7 Arguments d'une fonction

Les fonctions *Python* :

- acceptent des valeurs par défaut,
- permettent de retourner plusieurs valeurs,
- peuvent accepter un nombre indéterminé d'arguments (non nommés au départ)
- permettent de nommer les arguments assignés ou de fournir des arguments complémentaires nommés. [2]

Une fonction peut s'exprimer, avec au choix, les arguments suivants :

```
def ma_fonction( argument, argument_optionel = -1, *args, **kwargs ):
```

- Si on ne précise rien, l'argument est un argument obligatoire : oublier sa valeur lors de l'appel de la fonction provoquera une erreur.
- Un argument optionel est un argument avec une valeur par défaut : si l'on ne précise pas sa valeur lors de l'appel, c'est la valeur par défaut qui sera utilisée.
- **args* est une liste d'arguments non nommés. Ils sont ajoutés en en fin d'appel de fonction. Un exemple typique est la définition d'une fonction *somme* acceptant un nombre illimité de termes.

Exemple de valeurs par défaut aux arguments d'une fonction :

Programme *norme.py* :

```
1  # -*- coding: utf-8 -*-
2
3  def norme(x, y=0, z=0):
4      return (x**2+y**2+z**2)**0.5
```

Ainsi, des appels à `norme(0,4,3)`, à `norme(4,3)`, à `norme(-5)`, à `norme(4,z=3)`, ou à `norme(0,z=4,y=3)` renvoient 5.0.
Un appel à `norme(y=4, z=3)` provoque une erreur : `x` n'a pas de valeur.

$\Rightarrow$ **Activité 6. 33**

Expliquez les résultats du programme suivant :

Programme *foncarg.py* :

```
1  #-*- coding: utf-8 -*-
2
3  def fonc_arg1(a=1,b=1,c=1):
4      resultat=a+b+c
5      return resultat
6
7  print(fonc_arg1(3,5,7))
8  print(fonc_arg1(3,6))
9  print(fonc_arg1())
```

2. Cette dernière possibilité ne sera pas étudiée.

Informatique CPGE TSI - Etablissement Saint Joseph - LaSalle

```
10
11 def fonc_arg2(a,b,c):
12     resultat=a+b+c
13     return resultat
14
15 print(fonc_arg2(3,5,7))
16 print(fonc_arg2(3,6))
```

Résultat :

```
15
10
3
15
Traceback (most recent call last):
  File "foncarg.py", line 17, in <module>
    print(fonc_arg2(3,6))
TypeError: fonc_arg2() missing 1 required positional argument: 'c'
```

6.6.8 Fonction avec nombre indéterminé d'arguments

Si cela peu paraître inutile, l'opportunité d'ajouter un nombre illimité de paramètres en fin de fonction peu s'avérer terriblement utile pour des fonctions de traitement.
Pour les puristes, il est certes possible de concevoir sa fonction pour imposer un argument de type liste mais cela peut alourdir inutilement l'utilisation de la syntaxe.

Voici l'exemple d'une fonction *somme_inf.py* capable de sommer un nombre illimité d'arguments :

```
1 #-*- coding: utf-8 -*-
2
3 def sommeinf(*liste_arguments):
4     resultat=0
5     for terme in liste_arguments:
6         resultat+=terme
7     return resultat
8
9 print("somme = ",sommeinf(10,5,3,98,65,34))
```

6.7 Listes en compréhension

Les listes en compréhension permettent de créer des listes de façon concise et très élégante.

6.7.1 Copie de liste

La variable *el* (comme *élément*) va parcourir la liste *maliste* :

```
>>> maliste=[1,2,3,4,5]
>>> [el for el in maliste]
[1, 2, 3, 4, 5]
```


Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

6.7.2 Filtrage par test

En utilisant l'instruction if, qui teste si la condition est vraie, on peut filtrer les éléments de la liste :

```
>>> [el for el in maliste if el>3]
[4, 5]
```

6.7.3 Filtrage par test de fonction

Les tests peuvent être sous forme de fonctions :

```
>>> def pair(nombre):
...     return nombre % 2 == 0
...
>>> [el for el in maliste if pair(el)]
[2, 4]
```

6.7.4 Application d'une fonction

On peut appliquer une fonction (définie préalablement) aux éléments d'une liste :

```
>>> [el**2 for el in maliste if pair(el)]
[4, 16]
```

⇒ **Activité 6. 34**

1. Écrivez une seule ligne de 43 caractères, contenant l'instruction if, et qui affiche une liste des carrés des nombres divisibles par 5 compris entre 0 et 100 inclus. (L'instruction print() et les espaces sont comptés dans les 43 caractères).
2. Vérifiez à l'aide d'une autre ligne que la précédente compte bien 45 caractères.

6.8 Fonctions astucieuses

Certaines fonctions comme zip, map et lambda sont illustrées dans les programmes suivants :

comp1.py :

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Tue May 30 08:29:19 2017
4
5  @author: beutier.m
6  """
7
8  # On peut utiliser la fonction enumerate qui retourne
9  # chaque élément et sa position dans l'ensemble :
10
11 ma_liste = ["zéro", 1, "deux", 3, 4, "Five"]
12 for i,el in enumerate(ma_liste):
```


Informatique CPGE TSI - - Etablissement Joseph - LaSalle

```
13      print(i, el)
14
15  # Pour faire la somme de deux listes terme à terme, on peut écrire :
16  liste_1 = [4, 5, 6]
17  liste_2 = [3, 10, 11]
18  s = 0
19  for i in range(0,len(liste_1)) :
20      s = s + liste_1[i] + liste_2[i]
21  print(s)
22  # Ou utiliser la fonction zip :
23  liste_1 = [4, 5, 6]
24  liste_2 = [3, 10, 11]
25  s = 0
26  for el1,el2 in zip(liste_1,liste_2) :
27      s = s + el1 + el2
28  print(s)
29
30  # Il est possible d'éviter une fonction pour éviter d'écrire
31  # une boucle avec la fonction map.
32  # Elle applique une fonction à chaque élément d'un ensemble.
33  def fonction (x) :
34      return x ** 2
35  liste_3 = [3, 7, 9]
36  liste_4 = map (fonction, liste_3)
37  print (liste_4)
38  print (list(liste_4))
```

comp2.py :

```
1   # -*- coding: utf-8 -*-
2   """
3   Created on Tue May 30 08:29:19 2017
4
5   @author: beutier.m
6   """
7
8   # Le mot-clé lambda permet de définir des fonctions au sein d'une
                                         expression.
9   def fonction (x):
10      return x**3
11  liste_1 = [3, 7, 9]
12  liste_2 = map (fonction, liste_1)
13  print (list(liste_2))
14  # On peut l'écrire :
15  liste_1 = [3, 7, 9]
16  liste_2 = map (lambda x : x**3, liste_1)
17  print (list(liste_2))
18
19  # Et si on veut ajouter un paramètre à la fonction lambda :
20  liste_1 = [3, 7, 9]
21  k = 3
22  liste_2 = map (lambda x, y=k : x**k, liste_1)
23  print (list(liste_2))
24
25  # Fonction eval()
26  a = 39
27  b = 2
28  for op in ["+","-","*","/","//","%","|","&","or","and","==","<",">","<<",">
                                         >","is"]:
29      print(str(a)+" "+op+" "+str(b),"->", eval(str(a)+" "+op+" "+str(b)))
30
31  print("39 : \t\t",format(a,"b").rjust(8,"0"))
32  print("2 : \t\t",format(b,"b").rjust(8,"0"))
```


Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

```
33 print("39 + 2 : \t",format(41,"b").rjust(8,"0"))
34 print("39 - 2 : \t",format(37,"b").rjust(8,"0"))
35 print("39 or 2 : \t",format(39,"b").rjust(8,"0"))
36 print("39 and 2 : \t",format(2,"b").rjust(8,"0"))
37 print("39 < 2 : \t",format(156,"b").rjust(8,"0"), " : décalage 2 bits vers
                                                    la gauche")
38 print("39 >> 2 : \t",format(9,"b").rjust(8,"0"), " : décalage de 2 bits
                                                    vers la droite")
```

À vous d'en faire bon usage, même si elle ne sont pas au programme...

6.9 Gestion des erreurs

Parfois, une erreur est générée par *Python* et cela se termine par un message d'insultes ainsi que l'arrêt du programme.

Pour éviter cela, il y a quatre mots-clés vont servir à gérer les erreurs.

`try` permet "de tester" un bout de code. Si une erreur est rencontrée, on cesse d'interpréter le code et on passe aux `except`, qui permettent d'agir en fonction de l'erreur qui s'est produite. C'est pourquoi `try` doit toujours être suivi d'au moins un `except` ou d'un `finally`.

Exemple avec le code suivant :

```
try:
      reponse = int(input('Entrez un nombre entier : '))
except:
      print("Une erreur est survenue")
else:
      print("Le nombre est ",reponse)
```

- Si un nombre entier est saisi, il est ensuite affiché,
- Si un flottant ou une chaîne ou autre est entré, le message "Une erreur est survenue" est affichée.

Différents types d'erreurs existent. On peut citer :

- `ZeroDivisionError`
- `NameError`
- `TypeError`
- `ValueError`
- `IOError`
- `IndexError`
- `RuntimeError`
- `ImportError`
- `TabError`

En règle générale, l'instruction `try` fonctionne ainsi :

- D'abord, la clause d'essai (clause `try:` les instructions entre les mots-clés `try` et `except`) est exécutée,
- S'il ne se produit pas d'exception, la clause d'exception (clause `except`) est ignorée, et l'exécution du `try` est alors terminée,
- Si une exception se produit à un moment de l'exécution de la clause d'essai, le reste de la clause `try` est ignoré. Ensuite, si son type correspond à l'exception donnée après le mot-clé `except`, la clause `except` est exécutée, puis l'exécution reprend après l'instruction `try`,
- Si une exception se produit qui ne correspond pas à l'exception donnée dans la clause `except`, elle est renvoyée aux instructions `try` extérieures ; s'il n'y a pas de prise en charge prévue, il s'agit alors d'une exception non gérée et l'exécution est arrêtée avec un message d'insultes !
- `else` permet d'exécuter la suite du programme dans le cas où aucune exception ne survient,

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle

- `finally` applique toujours le bloc, qu'une exception soit levée ou non.

Une instruction `try` peut avoir plusieurs clauses d'exception, de façon à définir des gestionnaires d'exception différents pour des exceptions différentes.

Ainsi, il est possible d'avoir :

```
reponse = input('Entrez un nombre entier : ')

try:
      inverse = 1/float(reponse)
except ValueError:
      print(reponse,"n'est pas un nombre !")
except ZeroDivisionError:
      print("Division par zéro impossible !")
else:
      print("Son inverse est ",inverse)
```

On peut également indiquer plusieurs erreurs entre parenthèses :

```
reponse = input('Entrez un nombre entier : ')

try:
      inverse = 1/float(reponse)
except (ValueError,ZeroDivisionError):
      print("Un problème est survenu")
else:
      print("Son inverse est ",inverse)
```

Il est également possible de nommer et d'utiliser une erreur.

Un exemple pratique en utilisant `except...as` dans le programme *try_except.py* :

```
1  # -*- coding: utf-8 -*-
2
3  from math import sin
4  for x in range(-3, 4):
5      try:
6          print("\nx = {:.1f} donne\
7           sin(x)/x = {:.3f}".format(x,sin(x)/x))
8      except ZeroDivisionError as erreur: # exception
9          print("\nx = 0.0 donne\
10          sin(x)/x = 1.000") # exception pour x = 0
11         print("Cette erreur a été évitée : ",erreur)
12
13 for x in range(-3, 4):
14     print("\nx = {:.1f} donne\
15     sin(x)/x = {:.3f}".format(x,sin(x)/x))
```

et le résultat :

```
x = -3.0 donne         sin(x)/x = 0.047

x = -2.0 donne         sin(x)/x = 0.455

x = -1.0 donne         sin(x)/x = 0.841

x = 0.0 donne         sin(x)/x = 1.000
Cette erreur a été évitée :  float division by zero

x = 1.0 donne         sin(x)/x = 0.841

x = 2.0 donne         sin(x)/x = 0.455

x = 3.0 donne         sin(x)/x = 0.047
```

Dans certains cas, on peut être amené à créer des exceptions personnalisées. L'instruction `raise` peut être utilisée à ce propos. On peut également utiliser l'instruction `assert` amis ce ne sera pas détaillé ici.

Informatique CPGE TSI - Établissement Saint Joseph - LaSalle