Table des matières

1 Variables et types de données

Un langage de programmation permet de traiter des expressions, des variables et des constantes. Nous allons présenter les différents types d'objets que nous aurons à manipuler.

1.1 Variables et affectations

Définition : En informatique, une **variable** est un raccourci vers une zone mémoire où est stockée une information, par exemple un nombre.

En Python, la commande qui permet **d'affecter une valeur à une variable** est :

>>>nomdelavariable = valeur

Exemple :
```
>>> x = 8          La valeur 8 est affectée à la variable x.
>>> x              On demande la valeur de x.
8
```

Remarques : i) Les espaces dans l'expression $x = 8$ sont facultatives, c'est une convention de programmation. La commande $x = 8$ donne le même résultat.

ii) En Python, le nom de la variable doit commencer par une lettre. Python fait la différence entre une majuscule et une minuscule.

iii) On veillera à choisir un nom évocateur et qui ne correspond pas à un mot réservé du langage.

iv) Le symbole = ne correspond donc pas à une égalité mais à une affectation.

• La variable a vocation à être utilisée par la suite, comme dans l'exemple suivant :

Exemple :
```
>>> x = 3
>>> 2 * x + 1
7
```

• Dans de nombreux programmes, on sera amené à incrémenter la valeur d'une variable. On dit que Python autorise la réaffectation.

Exemple :
```
>>> x = 3
>>> x = x + 1
>>> x
4
```

• On peut effectuer des affectations simultanées.

Exemple :
```
>>> a, b = 3, 4.5
>>> a
3
>>> b
4.5
```

• On peut effacer le contenu d'une variable à l'aide de la fonction 'del'.

Exemple :
```
>>> a = 496
>>> del(a)          La variable a est réinitialisée.
```

1.2 Types des données

Quand on définit une variable en Python, il n'est pas utile d'en préciser son **type**, on parle de typage dynamique. En effet, pour chaque objet manipulé, le langage lui associe automatiquement une **classe** que l'on obtient avec la fonction **type**.

Exemple :
```
>>> type(123)
< class 'int' >
```

Remarque : On ignorera pour l'instant ce mot **class** qui fait référence au fait que Python est un langage orienté objet.

Voyons à présent les différents types de données que vous allez manipuler cette année.

1.2.1 Les nombres entiers

Le type **int** (integer) est celui des nombres entiers. Ils sont représentés de façon exacte et sans limite de taille.

Les opérations usuelles sur les entiers sont bien entendu utilisables en Python : $+$, $-$, $*$, $/$. On fera attention à la puissance qui se note $**$.

Exemple : `>>> 2 ** 12`
`4096`

Les règles usuelles de priorité sont reconnues en Python.

1.2.2 Les nombres flottants

Le type **float** (floating-point number) est celui des nombres réels. On utilise le symbole . à la place de la virgule.

Exemple : `>>> type(7.3)`
`<class 'float'>`

Remarque : La façon dont Python stocke et utilise les nombres à virgule sera étudiée plus tard dans l'année. On verra notamment le problème des erreurs d'arrondi.

- Les opérateurs $+$, $-$, $*$, $/$,$**$ sont également définis sur les nombres flottants.
- Lors d'une opération entre un flottant et un entier, l'entier est converti en flottant.

1.2.3 Les booléens

Le type **bool** (boolean) est celui est des propositions logiques. Il y a deux valeurs possibles : **True** ou **False**.

Exemple : `>>> 3 < 4`
`True`

- Les booléens servent à représenter le résultat de l'évaluation d'expressions logiques. Les principaux connecteurs logiques sont :

`==`	pour tester l'égalité	`!=`	différent de	`<`	inférieur strict
`<=`	inférieur ou égal	`>`	supérieur strict	`>=`	supérieur ou égal

Exemple : `>>> x = (3 == 2 + 2)`
`>>> x`
`False`

- Les principales opérations autorisées entre booléens sont : **and**, **or**, **not**. On rappelle que si P et Q sont deux expressions logiques, on a les règles d'évaluation suivantes :

P	Q	not(P)	P or Q	P and Q
True	True	False	True	True
True	False	False	True	False
False	True	True	True	False
False	False	True	False	False

Exemple : `>>> (1 > 2) or (2 == 4 - 2)`
`True`

1.2.4 Les chaînes de caractères

Le type **str** (string) est celui des chaînes de caractères. Ce type permet de représenter les textes.

Une chaîne de caractères se note entre apostrophes.

Exemple : `>>> t = 'helloworld'`
`>>> t`
`'helloworld'`
`>>> print(t)`

helloworld

Il y a de nombreuses opérations autorisées sur les chaînes de caractères :

Exemple : >>> a = 'hello'
>>> b = 'world'
>>> $a + b$
'helloworld'
>>> $a * 3$
'hellohellohello'
>>> $a[1]$
'e'

1.2.5 Les listes

Le type **list** est celui des suites d'éléments quelconques séparés par une virgule et encadrés par des crochets.

Exemple : >>> L = [7, 8, 3.2, 19, 'hello']
>>> $type(L)$
< class 'list' >
>>> $L[1]$
8

• Comme pour les chaînes de caractères et comme pour tous les objets indexables en Python, il faut avoir en tête que :

les éléments d'une liste sont numérotés à partir de 0.

• On peut extraire plusieurs éléments d'une liste simultanément :

Exemple : >>> L = [7, 8, 3.2, 19, 'hello']
>>> $L[1 : 3]$
[8, 3.2]

• En Python les listes sont des objets mutables, c'est-à-dire que l'on peut modifier un ou plusieurs éléments de la liste :

Exemple : >>> L = [3, 1, 4, 1, 5, 9]
>>> $L[2]$ = 7
>>> L
[3, 1, 7, 1, 5, 9]

Remarques : i) Il existe des conversions possibles entre les différents types de données.

Exemples : >>> $int(3.0)$
3
>>> $str(7.4)$
'7.4'

ii) Cette liste de types de données n'est pas exhaustive, vous aurez l'occasion de travailler durant l'année avec d'autres objets : tuples, dictionnaires, nombres complexes, matrices...

2 Introduction à l'algorithmique

Définition : Un **algorithme** est une suite d'instructions qui, exécutée correctement, conduit à un résultat donné.

Exemples : i) L'algorithme d'Euclide qui prend en entrée deux entiers relatifs et fournit en sortie le p.g.c.d des deux nombres.
ii) Algorithme de tri d'une liste qui prend en entrée une liste de nombres et renvoie la liste triée dans l'ordre croissant.

• Les critères de qualité attendus pour un algorithme sont : - lisibilité
- exactitude
- extensibilité
- portabilité
- efficience
- arrêt.

Tous les algorithmes peuvent être exprimés à l'aide de cinq instructions :

- ▶ **la déclaration** de variables,
- ▶ **l'affectation** de variables,
- ▶ **la séquence** qui exécute deux instructions l'une à la suite de l'autre,
- ▶ **le test**, ou instruction conditionnelle, qui sert à n'exécuter une instruction que dans certains états,
- ▶ **la boucle**, qui exécute plusieurs fois la même instruction dans le programme.

2.1 Entrées et sorties

Définition : Dans un langage de programmation, on appelle **entrées et sorties** les instructions qui interrompent le flot d'éxécution du programme pour communiquer avec l'utilisateur, donnant un aspect interactif au programme.

• Il existe une instruction particulière, **input**, qui attend que l'utilisateur tape quelque chose au clavier et qui prend pour valeur la chaîne de caractères correspondante. On l'utilise très souvent sous la forme variable = input(), de sorte que la variable va contenir ce qui est tapé au clavier.

Exemple : `>>> s = int(input('Quel est votre nombre préféré ?'))`
`Quel est votre nombre préféré ? 73`

• L'instruction **print** affiche à l'écran la ou les expressions qui lui sont données en argument. Elle ne modifie pas les contenus des variables mais seulement l'aspect de l'écran. C'est en particulier utile pour afficher un résultat issu de l'exécution d'un programme.

Exemple : `>>> s = 73`
`>>> print('La variable s contient', s, 'et rien de plus')`
`La variable s contient 73 et rien de plus`

2.2 Instructions conditionnelles

2.2.1 Test simple

Une instruction conditionnelle n'est exécutée que si une condition donnée est vérifiée. Pour traduire cela, on utilise l'instruction **if** en Python qui a la syntaxe suivante :

```
if condition :
    instruction1
    instruction2
    ...
suite des instructions
```

Exemple :
```
if x % 2 == 1 :
    x = x + 1
```

Remarques : i) Pour identifier sans ambiguité les instructions appartenant au bloc du **if**, il est nécessaire de les indenter : c'est une obligation en Python. Une indentation est constituée de quatre espaces. En TP, vous remarquerez que Pyzo propose automatiquement une indentation.

ii) Une erreur fréquente est d'oublier les " :" qui suivent la condition.

ii) La condition doit être une variable booléenne constituée des connecteurs logiques que nous avons vus au chapitre précédent.

2.2.2 Test avec alternative

• Pour exprimer une condition du type "**si...alors...sinon**", on utilise la syntaxe suivante :

```
if condition :
    instructions
else :
    instructions
suite des instructions
```

Exemple :
```
if n % 2 == 0 :
    print('n est pair')
else :
    print('n est impair')
```

• Il est également possible de contracter un **else** suivi d'un **if** en **elif**. Cette syntaxe permet de distinguer facilement plusieurs cas.

Exemple :
```
if x < 0 :
    print('x est strictement négatif')
elif x < 3 :
    print('x est compris entre 0 et 3')
elif x < 5 :
    print('x est compris entre 3 et 5')
else :
    print('x dépasse 5')
```

2.2.3 Tests imbriqués

Il est possible de réaliser une instruction conditionnelle dans une instruction conditionnelle. Il faut alors ajouter une indentation.

Exemple :
```
if n % 2 == 0 :
    print('n est pair')
    if n % 4 == 0 :
        print('n est un multiple de 4')
    else :
        print('n n'est pas un multiple de 4')
else :
    print('n est impair')
```

2.3 Boucles

Définition : Une boucle est utilisée en programmation pour répéter plusieurs fois un bloc d'instructions.

2.3.1 Boucle inconditionnelle

• La boucle **for** est utilisée lorsque l'on connaît à l'avance le nombre d'itérations qu'il faut effectuer. Sa syntaxe est la suivante :

```
for variable in range(valeur initiale, valeur finale, pas) :
    bloc d'instructions à répéter
```

• La première fois que Python lit l'instruction **for**, la variable prend la valeur initiale, ceci même si elle avait une valeur auparavant. Le bloc d'instructions est ensuite effectué. À la fin du bloc, la variable est incrémentée du pas et le bloc est à nouveau effectué depuis le début. On arrête la boucle lorsque la valeur devrait dépasser la valeur finale.

Exemple :
```
for i in range(3,11,2):
    print(i)
```

Ce programme va afficher 3,5,7,9. Chaque nombre sera sur une ligne différente.

Remarques : i) La valeur finale n'est pas prise en compte dans la boucle.

ii) Après exécution de la boucle, la variable conserve la valeur qu'elle avait après la dernière exécution du bloc d'instructions.

iii) Les trois arguments du **range** doivent être des entiers. Le pas ne doit pas être nul.

iv) Pour donner des valeurs décroissantes à la variable, on peut utiliser un pas négatif.

v) Si **range** est utilisé avec deux arguments le pas vaut 1 par défaut.

vi) Si **range** est utilisé avec un seul argument, le pas vaut 1 et la valeur initiale vaut 0.

vii) On peut imbriquer des **for** et des **if** sans contrainte.

Exemple :
```
p = 1
for i in range(n):
    p = (i+1) * p
```

• Il est également possible de parcourir une liste ou une chaîne de caractères. Avec l'instruction suivante la variable prendra comme valeurs successives les éléments de la liste ou de la chaîne de caractères.

```
for variable in liste (ou chaîne) :
    bloc d'instructions à répéter
```

Exemple :
```
L = [3, 6, 10]; a = 0
for i in L:
    a = a + i
print(a)
```

Ce programme renvoie la somme des termes de la liste L.

Exemple :
```
c = 'truc'
for i in c:
    print(i)
```

Si l'on compile ce programme, il va afficher : t, r, u, c. Chaque lettre étant sur une ligne différente.

2.3.2 Boucle conditionnelle

Si l'on connaît à l'avance le nombre de répétitions à effectuer dans une boucle, on choisit une boucle **for**. À l'inverse, si la décision d'arrêter la boucle ne peut s'exprimer que par un test, c'est la boucle **while** qu'il faut choisir.

Lorsque l'on lit l'instruction **while**, la condition est évaluée. Si elle est fausse, on saute le bloc et on passe directement à la suite. Si elle est vraie, on effectue le bloc puis on revient sur la ligne **while**. La condition est à nouveau évaluée et deux cas se présentent à nouveau. Ceci jusqu'à ce que la condition évaluée soit fausse. La syntaxe de la boucle **while** est la suivante :

```
while condition :
    bloc d'instructions
```

Exemple :
```
n = int(input('Entrez un nombre entier positif : '))
p = 1
while p <= n:
    p = p * 2
print(p)
```

Ce programme affiche la plus petite puissance de 2 strictement supérieure à n.

3 Fonctions

3.1 Notion de fonction

3.1.1 Un exemple

Dans le chapitre précédent, nous avons appris à réaliser de nombreux programmes à l'aide de structures conditionnelles et itératives. Nos programmes recevaient les données de l'utilisateur via la commande **input** et affichaient un résultat avec la commande **print**.

Exemple : *Voici un programme qui calcule la factorielle d'un entier :*

```
n = int(input('Entrez un nombre entier positif : '))
p = 1
for k in range(1,n + 1):
    p = p * k
print(p)
```

• Avec cette façon de procéder, il est assez fastidieux de calculer la factorielle de plusieurs entiers puisque l'on doit relancer le programme à chaque fois. De plus, il est impossible de faire appel à ce programme à l'intérieur d'un autre programme.

• On aimerait donner un nom au programme que l'on vient de créer, par exemple **fact**, et pouvoir le réutiliser autant de fois que l'on souhaite ou encore l'inclure dans des calculs du type : $2 * fact(3) + 1$.

• La bonne façon de faire consiste à définir une fonction, ce qui se fait au moyen de la commande **def**. Reprenons l'exemple précédent :

Exemple :
```
def fact(n):          # Commande qui définit la fonction fact
    p = 1
    for k in range(1,n + 1):
        p = p * k
    return(p)
```

• Là aussi, il convient de prêter attention à l'indentation. Cependant Python vous proposera automatiquement les 4 espaces usuelles. L'instruction *return p* fonctionne aussi.

• Lorsque l'on exécute le fichier contenant cette fonction, Python crée une nouvelle variable appelée **fact** de type function. Cette variable contient le bloc d'instructions lui-même. À ce moment le bloc d'instruction n'est pas exécuté mais simplement enregistré et le programme passe à la suite.

• Il est ensuite possible de faire appel à la fonction **fact** ainsi créée autant de fois que l'on souhaite dans la console Python ou dans la suite du fichier contenant la fonction.

Exemple :
```
>>> fact(3)
6
>>> 2 * fact(4) - 1
47
>>> a = fact(5)
>>> a
120
```

3.1.2 Définition et vocabulaire

Définition : Une fonction est une suite d'instructions qui dépendent de paramètres.

• La variable n dans la définition de la fonction **fact** s'appelle le **paramètre** de la fonction. Ce paramètre prend la valeur transmise lors de l'appel. La valeur renvoyée par la fonction est celle qui suit la commande **return**.

• Il est tout à fait possible de définir une fonction prenant en entrée plusieurs paramètres.

Exemple :
```
from math import sqrt

def hypothenuse(x, y):
    z = sqrt(x * x + y * y)
    return(z)
```

Si l'on fait appel à cette fonction dans la console :

```
>>> hypothenuse(3, 4)
5
```

• Il est aussi possible de ne pas spécifier à l'avance le nombre de paramètres utilisés.

Exemple :
```
def produit(*facteurs):
    p = 1
    for i in facteurs:
        p = p * i
    return(p)
```

Python place ces paramètres dans un n-uplet.

• Il est également tout à fait possible de renvoyer plusieurs résultats : $return(x, y, z...)$.

• Le bloc d'instructions correspondant à la fonction peut contenir n'importe quelles instructions Python, par exemple :
- ▶ **if, for, while**...
- ▶ des appels à d'autres fonctions.
- ▶ des appels à la fonction elle-même.

Exemple :
```
def f(x):
    return(x * x)

def g(x):
    return(f(x) + 1)

>>> g(5)
26
```

Il n'est même pas nécessaire que la fonction f ait été définie avant la fonction g, il suffit que les deux fonctions aient été définies avant l'appel $g(5)$.

3.1.3 La commande return

• On peut placer des instructions **return** n'importe où dans le corps d'une fonction. La première fois qu'une instruction **return** est lue, l'exécution de la fonction est interrompue et la valeur indiquée par **return** est renvoyée par la fonction.

Exemple : La fonction suivante teste si un entier est un carré :

```
def testcarre(n):
    for k in range(n+1):
        if k * k == n:
            return(True)      # Si on lit cette ligne l'exécution est stoppée et renvoie True
    return(False)             # Si on arrive là, c'est que n n'est pas un carré
```

3.1.4 Print ou return ?

• On peut être amené à écrire une fonction provoquant une certain action, par exemple un affichage, mais qui ne renvoie aucune valeur. Dans ce cas, on ne met aucune commande **return**. Une fonction qui ne renvoie pas de valeur est aussi appelée une **procédure**.

• Il ne faut pas confondre les instructions **print** et **return**. On utilise **print** pour provoquer un affichage et **return** pour renvoyer une valeur. Pour mettre en lumière la différence, considérons les deux fonctions suivantes :

Exemple :
```
def carre1(x):
    return(x * x)
```
```
def carre2(x):
    print(x * x)
```

La première fonction renvoie x^2 lorsqu'elle reçoit un nombre x et on peut l'inclure dans un calcul. Autrement dit $carre1(5)$ est une valeur, ici 25. La seconde fonction se contente d'afficher le résultat 25 mais la fonction carre2 est inutilisable dans un calcul car elle ne renvoie pas de valeur.

• Bien entendu, il est possible d'inclure dans une fonction des commandes **print** et **return** à la fois. Toutes les commandes **print** exécutées avant la lecture d'un **return** vont provoquer un affichage.

3.1.5 Concevoir une fonction

• Quand on conçoit une fonction, il est préférable de lui donner un nom explicite car elle est susceptible d'apparaître à plusieurs endroits du programme.

• Il est possible de documenter une fonction en expliquant à un futur utilisateur les valeurs des paramètres possibles, le résultat rendu, etc...Python propose un mécanisme pour associer une documentation à une fonction sous la forme d'une chaîne de caractères placée au début du corps de la fonction.

Exemple :
```
def fact(n):
    "Cette fonction calcule la factorielle d'un entier naturel à l'aide d'une boucle"
    p = 1
    for k in range(1, n + 1):
        p = p * k
    return(p)
```

• Le plus intéressant est que vous pouvez ensuite avoir accès à la documentation de la fonction en tapant dans la console : $help(fact)$.

3.2 Variables locales et variables globales.

3.2.1 Variables locales

• L'utilisation des fonctions facilite la conception des programmes et améliore leur lisibilité. Un long programme d'une centaine de lignes d'un seul bloc est illisible. Le même programme structuré en plusieurs fonctions documentées est plus facile à appréhender. De plus, chaque fonction peut être testée séparement, ce qui facilite grandement la recherche d'erreurs. Lorsque l'on écrit plusieurs fonctions dans un même fichier, on va souvent être amené à utiliser plusieurs fois une même variable pour des usages distincts, par exemple l'indice i d'une boucle. On aimerait que ces différentes utilisations n'interfèrent pas entre elles ou avec des variables déjà existantes en dehors de la fonction. Python sait faire cette distinction.

Définition : Une variable **locale** est un variable dont la portée est limitée au corps d'une fonction.

• Toute variable créée ou modifiée dans une fonction est automatiquement considérée comme une variable locale par Python. Ce n'est pas le cas de tous les langages de programmation puisque dans certains d'entre eux, il faut explicitement déclarer les variables locales.

Exemple :
```
x = 3                    # x prend la valeur 3 à l'extérieur de la fonction.
def exemple():           # Une fonction sans paramètre
    x = 5
    print(x)

>>> exemple()
5
>>> x
3
```

*Durant l'exécution de exemple(), une nouvelle variable x est automatiquement créée, appelée **variable locale**, pour ne pas écraser l'ancienne variable x qui existait déjà en dehors de la fonction, appelée **variable globale**. La portée de la variable locale est limitée à l'exécution de l'appel de la fonction.*

• Les paramètres d'une fonction sont automatiquement considérés comme des variables locales.

• Le mécanisme des variables locales est une sécurité : on peut introduire des variables auxiliaires dans le corps d'une fonction sans se préoccuper de savoir s'il existe déjà une variable du même nom définie à l'extérieur de la fonction.

3.2.2 Lecture d'une variable globale

• Dans une fonction, on peut être amené à utiliser une variable existante en dehors.

Exemple :
```
pi = 3.1416
def aire(R):
    return(pi * R * R)          # Ici la variable pi est globale.
```

• Une variable dont le contenu est lu mais non modifié par la fonction est automatiquement considérée comme une variable globale.

3.2.3 Modification d'une variable globale

• Il est possible de forcer Python à modifier, au sein d'une fonction, une variable globale à l'aide de la déclaration **global**.

Exemple :

```
a = 3

def exemple2():
    global a
    a = 5

>>> exemple()
>>> a
5
```

3.2.4 Fonctions locales

• Une fonction est une variable comme une autre, elle peut donc jouer le rôle d'une variable locale. Il est possible de définir une fonction à l'intérieur d'une autre fonction.

Exemple :

```
def g(x):
    def f(x):
        return(x * x)
    return(f(x) + 1)
```

Ce que l'on peut également écrire :

```
def f(x):
    return(x * x)

def g(x):
    return(f(x) + 1)
```

Dans le second cas, la fonction f est globale donc accessible dans tout le programme. Dans le premier cas, la fonction f est créée à chaque appel de g puis détruite ensuite, elle est inutilisable en dehors de g.

3.3 Mécanismes avancés

3.3.1 Fonctions de bibliothèque

• Tous les langages de programmation proposent des fonctions toutes faites pour la plupart des besoins courants et écrites par les concepteurs du langage. En Python, les différentes fonctions sont organisées en modules. Il existe de très nombreux modules dédiés à des thématiques très variées. Les modules les plus utiles pour un élève de classe préparatoire sont :

- ▶ **math** qui contient toutes les fonctions et constantes utilisées en analyse.
- ▶ **random** qui fournit des nombres pseudo-aléatoires.
- ▶ **numpy, scipy, matplolib** qui fournissent des outils variés pour le calcul scientifique et la représentation graphique.
- ▶ **Tkinter** qui fournit une interface graphique.
- ▶ **time** qui permet d'accéder aux fonctions gérant le temps.

*Exemple : Vous avez déjà utilisé la fonction **sqrt** du module **math**. La commande pour l'importer est :*

```
from math import sqrt
```

• Il est également possible de charger toutes les fonctions du module **math** directement avec la commande : *from math import ∗*
Cette méthode a un inconvénient : il existe plusieurs fonctions avec le même nom issues de modules différents. Par exemple les modules **math** et **numpy** possèdent tous deux une fonction **sqrt**. On peut les distinguer ainsi : $math.sqrt(3)$ et $numpy.sqrt([4,5])$.

• Vous découvrirez au cours de l'année les nombreuses fonctions présentes dans ces modules.

• Il est possible d'obtenir de l'aide sur l'utilisation d'une fonction. Par exemple, la commande : $help(asin)$ vous fournira des explications concernant l'utilisation de la fonction Arcsinus. Bien entendu, l'aide en ligne de Python peut également vous renseigner sur les différentes fonctions présentes dans un module.

3.3.2 Utiliser une fonction en tant que paramètre

• En Python, une fonction est une variable comme une autre, c'est-à-dire qu'il est possible de la passer en paramètre d'une autre fonction.

Exemple :
```
def somme(f, n) :
    s = 0
    for i in range(1, n + 1) :
        s = s + f(i)
    return(s)

def carre(x) :
    return(x * x)
```

Dans la console, on peut alors effectuer :

```
>>> somme(carre, 10)
385
```

• Dans l'exemple précédent, il est possible d'éviter de nommer la fonction **carre** puisqu'elle est réduite à une simple expression. L'écriture : *lambda x : x * x* désigne, dans le langage de Python, la fonction $x \mapsto x^2$. On a alors :

```
>>> somme(lambda x : x * x, 10)
385
```

Remarque : La commande **lambda** permet donc d'abréger la syntaxe de définition d'une fonction. Par exemple, on peut écrire dans la console :

```
>>> f = lambda a, b, c : a + b + c
>>> f(1, 2, 4)
7
```

• De même que l'on peut passer une fonction en paramètre, il est possible de renvoyer une fonction comme résultat avec la commande **return**.

3.3.3 Fonctions partielles

• Les fonctions considérées jusqu'à présent sont des **fonctions totales**, c'est-à-dire qu'elles renvoient toujours un résultat quelle que soit la valeur de leurs paramètres. Il existe aussi des fonctions dites **partielles** parce qu'elles ne renvoient pas un résultat pour toutes les valeurs possibles des paramètres. La fonction suivante est partielle :

Exemple :
```
def divise(x, y)
    return x // y
```

Tout appel à cette fonction avec un deuxième argument égal à 0 *va provoquer un message d'erreur.*

• Pour définir proprement une fonction partielle, il est préférable d'inclure une **précondition** au début du corps de la fonction à l'aide d'une instruction **assert**. Il est possible de lister plusieurs préconditions.

Exemple :
```
def divise(x, y)
    assert y != 0, 'division par 0 impossible'
    return x // y
```

À présent, cette fonction informe l'utilisateur avec le message d'erreur choisi.

3.3.4 Un mot sur la récursivité.

Définition : Une fonction récursive est une fonction récursive.

• Cette définition illustre bien le concept de récursivité. Plus clairement, une fonction récursive est une fonction qui fait appel à elle-même. Nous étudierons plus en détail les fonctions récursive cette année. Donnons simplement un exemple très classique :

Exemple :
```
def factorielle(n) :
    if n == 0 :
        return(1)
```

else :
    $return(n * factorielle(n-1))$

• Il important de noter qu'une telle fonction ne se terminera pas avec un argument n négatif. On pourrait améliorer le programme en ajoutant la condition : $assert\ n >= 0$.

• Le langage Python limite le nombre d'appels récursifs à 1000 et il existe certaines fonctions récursives dont il n'est pas aisé de démontrer la terminaison.

4 Compléments sur les listes

4.1 Programmation objet

• Le langage Python appartient à la famille des langages dits **orientés objet**. Cela signifie que chaque type de données (appelé aussi une **classe**) possède des éléments qui le composent (appelés **attributs**) mais également des fonctions ou des programmes qui agissent sur les objets de la classe (on parle de **méthodes**).

Exemple : *Nous avons vu dans le chapitre* 1 *la classe des listes. Dans cette classe, une liste est un objet. Parmi les attributs d'un objet de la classe liste, on peut citer ses éléments. Nous allons étudier dans ce chapitre les méthodes de la classe liste, on peut donner un premier exemple :*

```
>>> L = [1, 2, 3]
>>> L.reverse()
>>> L
>>> [3, 2, 1]
```

• Il est tout à fait possible de définir vos propres classes en Python. Par exemple, si vous souhaitez manipuler des polynômes, vous pouvez créer une classe polynôme. Il faudra détailler la création d'un polynôme, en le voyant par exemple comme une liste de coefficients. Vous pourrez ensuite définir vos propres méthodes qui vont agir sur les éléments de la classe polynôme : degré, dérivée...

4.2 Méthodes sur les listes

• Voyons un exemple :

Exemple : *On définit une liste :*

```
>>> L = [1, 6, 8]
```

On peut avoir accès à l'adresse mémoire dans laquelle est stockée L :

```
>>> id(L)
4641165896
```

On souhaite ajouter l'entier 5 *à la fin de cette liste, nous avons vu que l'opération* + *effectue cette concaténation :*

```
>>> L = L + [5]
>>> L
[1, 6, 8, 5]
>>> id(L)
4641132128
```

On constate que l'adresse mémoire a changé lors de l'opération. L'interpréteur Python a effectué une copie de la première liste, y a ajouté l'élément 5 *et a réaffecté la variable L. Cette façon de faire est coûteuse au niveau de la gestion de la mémoire. La méthode* ***append()*** *va effectuer cette concaténation sans changer l'adresse mémoire. On dit que la modification a lieu sur place.*

```
>>> L = [1, 6, 8]
>>> id(L)
4641165896
>>> L.append(5)
>>> L; id(L)
[1,6,8,5]
4641165896
```

• Une méthode va permettre de modifier directement un objet sans avoir à en faire une copie. Ce phénomène est appelé **encapsulation** des données et des méthodes.

• Outre le gain en place mémoire, l'utilisation de la méthode **append()** au lieu de l'opération + va également induire un gain de temps.

• Le choix pédagogique des concepteurs est de ne pas vous faire apprendre un liste exhaustive de méthodes, les seules commandes au programme sont :

Exemples :
- $L.append(x)$: ajoute l'élément x à la fin de la liste L.
- $L.pop()$: supprime l'élément en dernière position et le renvoie.
- $len(L)$: renvoie le nombre d'éléments de la liste L.

• Il existe de nombreuses autres commandes permettant de manipuler les listes, même si elles ne sont pas exigibles, on peut citer :
- $L.count(x)$: renvoie le nombre d'occurrences de x.
- $L.sort()$: modifie la liste en la triant.
- $L.reverse()$: modifie la liste en inversant l'ordre des éléments.

4.3 Listes en compréhension

• Pour créer des listes, Python fournit une facilité syntaxique particulièrement agréable car proche des notations utilisées en mathématiques. Cela permet de générer des listes d'une manière très concise sans avoir à utiliser de boucles. On parle de listes définies par **compréhension**.

Exemples :
```
>>> L = [2, 4, 6, 8, 10]
>>> [3 * x for x in L]
[6, 12, 18, 24, 30]
```

```
>>> [[x, x **3] for x in L]
[[2, 8], [4, 64], [6, 216], [8, 512], [10, 1000]]
```

On peut également filtrer les éléments en ajoutant une condition.

```
>>> [3 * x for x in L if x > 5]
[18, 24, 30]
```

On peut imbriquer des boucles for.

```
>>> M = range(3)
>>> [x * y for x in L for y in M]
[0, 2, 4, 0, 4, 8, 0, 6, 12, 0, 8, 16, 0, 10, 20]
```

Remarque : La façon la plus rapide de créer une liste est de la définir par compréhension. C'est plus efficace que l'utilisation d'une boucle for et de l'opérateur de concaténation ou que la méthode append(). Vous pouvez consulter l'exercice 3 du chapitre 4.

4.4 Copier une liste

• Pour comprendre la subtilité concernant la copie des listes en Python, considérons l'exemple suivant.

Exemples :
```
>>> L = [1, 2, 3]
>>> M = L
>>> L[1] = 17
>>> M
[1, 17, 3]
```

La liste M est modifiée lorsque L est modifiée. Lorsque l'on effectue M = L, on ne copie que les références des listes, on dit que L et M pointent vers le même objet liste. On peut d'ailleurs s'en assurer :

```
>>> id(L) == id(M)
true
```

Cela devient vite problématique dès lors que l'on souhaite travailler sur une liste en conservant les données initiales.

• Il y a plusieurs techniques pour créer une copie d'une liste indépendante de la liste initiale.

1)
```
>>> L = [1, 2, 3]
>>> M = list(L)
```

2)
```
>>> L = [1, 2, 3]
>>> M = L[:]
```

3)
```
>>> import copy as c
>>> L = [1, 2, 3]
>>> M = c.copy(L)
```

4)
```
>>> import copy as c
>>> L = [1, 2, 3]
>>> M = c.deepcopy(L)
```

Ces différentes instructions fonctionnent très bien lorsqu'il s'agit de cloner des listes de nombres par exemple. Par contre, les techniques 1, 2 et 3 ne permettront pas de copier indépendamment une liste de listes comme le montre l'exemple suivant.

Exemples :
```
>>> L = [1, 2, [3,4]]
>>> M = list(L)
>>> M[2][0] = 15
>>> M
[1, 2, [15,4]]
>>> L
[1, 2, [15,4]]
```

Le même problème se retrouve également lorsque l'on utilise les techniques 2 et 3. Dans ce cas, la seule façon de cloner correctement une liste de listes sera d'utiliser la technique 4 avec la commande *deepcopy*.

4.5 Les slices en Python

Le terme anglais de slice est associé à l'idée de découpage (une part de gâteau ou de pizza). En programmation et en Python en particulier, un slice permet le découpage de données séquentielles comme les chaines de caractères ou les listes. Voici quelques exemples qui illustrent les commandes existantes.

```
>>> L = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9]
>>> L[3 : 7]    ♯renvoie les éléments d'indices compris entre 3 (inclus) et 7 (exclu)
[1, 5, 9, 2]

>>> L[−5]    ♯un indice négatif permet d'accéder à la liste en partant de la fin
5
```

```
>>> L[3 : −5]
[1, 5, 9, 2, 6]

>>> L[6 : 50]     ♯les slices tolèrent les dépassements d'indices
[2, 6, 5, 3, 5, 8, 9]

>>> L[: 5]      ♯les 5 premiers
[3, 1, 4, 1, 5]

>>> L[−5 :]      ♯les 5 derniers
[5, 3, 5, 8, 9]

>>> L[5 :]      ♯tous sauf les 5 premiers
[9, 2, 6, 5, 3, 5, 8, 9]

>>> L[: −5]      ♯tous sauf les 5 derniers
[3, 1, 4, 1, 5, 9, 2, 6]

>>> L[:]      ♯tous les éléments
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9]
```

La syntaxe des slices admet une extension autorisant un troisième entier, cet entier désigne un pas. Ce pas peut être négatif mais pas nul.

```
>>> L[2 : 10 : 3]
[4, 9, 5]

>>> L[−5 : 2 : −2]
[5, 2, 5]
```

La syntaxe des slices permet aussi de modifier une liste suivant une tranche.

```
>>> L[2 : 8] = [0, 0]
>>> L
[3, 1, 0, 0, 5, 3, 5, 8, 9]
```

Remarque : Toutes ces syntaxes sont également valables sur les chaines de caractères.

5 Exercices

5.1 Énoncés

1 Prévoir le résultat des opérations suivantes :

a) >>> (1 < 2) or (3.4 > 5)
b) >>> (1 < 2) == (5 == 5)
c) >>> ((2 < 1) or (not(2 < 2))) and (not((2 >= 2) or (−1 > 3)))
d) >>> (1 == 1) + 7

2 Deviner le résultat de l'instruction suivante.

>>> L = $[5, 9, 'hello', 8.9, 4]$
>>> $L[2][2]$

3 Écrire un programme qui trouve le maximum de trois nombres entiers donnés par l'utilisateur.

4 Écrire un programme qui cherche un caractère dans une chaîne de caractères. On demandera à l'utilisateur de saisir la chaîne de caractères ainsi que le caractère à rechercher.

5 Écrire un programme qui détermine le rang du dernier terme strictement positif de la suite récurrence définie par $u_{n+1} = \frac{1}{2}u_n - 3n$, la valeur de u_0 étant donnée dans la variable u_0.

6 Écrire un programme qui demande à l'utilisateur de taper 'bonjour', et qui recommence indéfiniment jusqu'à ce que l'utilisateur ait obéi à la consigne.

7 Définir une fonction appelée *moyenne* qui, pour une liste donnée, renvoie la moyenne des valeurs de la liste.

8 On considère la fonction $f : x \mapsto x^4 - x^2 - 1$. On cherche à approcher le minimum de f sur l'intervalle $[0, 2]$.

a) Définir la fonction f.

b) Proposer des lignes de commande fournissant une liste de 1000 réels uniformément répartis dans l'intervalle $[0, 2]$, ainsi qu'une nouvelle liste correspondant aux images par la fonction f de ces 1000 réels.

c) Écrire une fonction qui propose une valeur approchée du minimum recherché.

d) Donner la valeur exacte de ce minimum.

9 a) Quelles sont les variables locales et globales de la fonction f ?

```
def f() :
    global a
    a = a + 1
    c = 2 * a
    return(a + b + c)
```

b) Qu'affiche le programme lorsque l'on effectue les instructions suivantes :

```
a, b, c = 3, 4, 5
print(f())
print(a)
print(b)
print(c)
```

10 Écrire une fonction qui prend comme argument deux fonctions f et g et qui renvoie la composée $f \circ g$.

11 Écrire une ligne de commande qui étant donné une liste d'entiers renvoie la même liste où l'on a enlevé chaque occurrence de l'élément 0.

12 Utiliser une liste en compréhension pour obtenir les diviseurs positifs d'un entier naturel n.

13 On reprend la liste $L = [3,\ 1,\ 4,\ 1,\ 5,\ 9,\ 2,\ 6,\ 5,\ 3,\ 5,\ 8,\ 9]$.

a) Écrire une ligne de commande pour insérer la liste $[0, 0, 0]$ entre le 9 et le 2 sans rien supprimer de la liste initiale.

b) Écrire une ligne de commande pour ajouter la liste $[0, 0, 0]$ à la fin de la liste initiale.

c) Écrire une ligne de commande pour effacer tous les éléments de la liste initiale correspondant aux indices 2, 3 et 4.

d) Écrire une ligne de commande pour extraire tous les éléments de la liste L de 3 en 3 à partir de l'indice 4.

e) Écrire une ligne de commande pour extraire tous les éléments d'indices impairs.

f) Écrire une ligne de commande qui renvoie la liste initiale dans l'ordre inverse.

5.2 Corrigés

1 On a :

a) True

b) True

c) False

d) On obtient 8 car 1 == 1 qui vaut *True* est interprété comme 1 lors de l'addition.

2 On obtient $'l'$ car la numérotation des listes et des chaines de caractères commence à 0.

3 On a :

```
a = int(input('a?'))
b = int(input('b?'))
c = int(input('c?'))
if (a >= b) and (a >= c):   #on regarde si a superieur a b et c
    print(a)
if (b >= a) and (b >= c):   #on regarde si b est superieur a a et c
    print(b)
else:                       #sinon c'est c le plus grand
    print(c)
```

4 On a :

```
chaine = input('Entrez votre texte : ')
compt = 0
car = input('caractere a rechercher : ')
for i in chaine:
    if car == i:
        compt = compt + 1     #on compte le nombre d'occurences du caractere
print(compt)
```

5 On a :

```
u0 = float(input('Entrez la valeur initiale : '))
u = u0
n = 0
while u > 0:
    u = 0.5 * u - 3 * n     #on calcule le terme suivant de la suite
    n = n + 1               #on incremente n
print(n - 1)
```

6 On a :

```
reponse = 'a'
while reponse != 'bonjour':
    reponse = input('tapez bonjour au clavier svp : ')
print('ok')
```

7 On a :

```
def moyenne(L):
    "renvoie la moyenne des elements de la liste L"
    S = 0 #on stocke la somme dans la variable S
    l = len(L) #le nombre d'elements de la liste L
    for i in range(l):
        S = S + L[i]
    return(S / l)
```

8 On a :

```
def f(x):
    return(x ** 4 - x ** 2 - 1)
```

b) Pas besoin de créer une fonction ici, on peut directement écrire :

```
def f(x):
    return(x ** 4 - x ** 2 - 1)

L = [2 * k / 1000 for k in range(0, 1000)]
M = [f(2 * k / 1000) for k in range(0, 1000)]
```

c) On va parcourir la liste M que l'on vient de créer pour en chercher le minimum.

```
def f(x):
    return(x ** 4 - x ** 2 - 1)

L = [2. * k / 1000 for k in range(0, 1000)]
M = [f(2. * k / 1000) for k in range(0, 1000)]

MM = [f(k) for k in L]

def min():
    "renvoie le minumum de la liste M precedente"
    m = M[0]
    for i in range(1, len(M)):
        if M[i] < m:
            m = M[i]
    return(m)
```

d) Une rapide étude de fonction nous montre que le minimum recherché est $-\frac{5}{4}$. La valeur approchée fournie par nos fonctions précédentes est -1.249998402304. Pour augmenter la précision, on pourrait choisir de prendre plus que 1000 réels.

9 a) La variable a est globale car déclarée comme telle. La variable b est globale car elle est simplement lue par la fonction mais non modifiée. La variable c est locale car elle est modifiée dans le corps de la fonction f.

b) $>>> print(f())$
16
$>>> print(a)$
4
$>>> print(b)$
4
$>>> print(c)$
5

10 On a :

```
def composee(f,g):
    "renvoie la fonction fog"
    def h(x):
        return(f(g(x)))
    return(h)
```

11 On a :

```
L = [x for x in L if x!=0]
```

12 On a :

```
n = 100; [d for d in range(1,n+1) if n % d == 0]
```

13 On a :

```
#1
L[6:6] = [0, 0, 0]

#2
L[len(L):] = [0, 0, 0]

#3
L[2:5] = []

#4
L[4::3]

#5
L[1::2]

#6
L[::-1]
```