Introduction

Le cours que nous proposons ici est conforme au programme d'informatique de tronc commun des classes préparatoires scientifiques [1]. Il peut bien sûr servir hors de ce contexte précis pour toute personne qui voudrait apprendre ou reprendre les bases de l'algorithmique avec Python.

Son organisation en trois parties (et quinze chapitres) est conforme à l'organisation suggérée par ce programme dont la logique nous convient parfaitement.

Nous avons fait le choix, dès la première partie et pour la quasi-totalité des algorithmes étudiés, de mettre en place soit dans le cours, soit dans les exercices, les preuves de programme et l'étude de la complexité. Dans les cas les plus délicats il sera possible de ne se focaliser dans un premier temps que sur la compréhension, la programmation et une vérification empirique de l'algorithme (les tests), pour revenir ensuite sur les aspects plus théoriques au moment de faire la synthèse et de prendre du recul sur ces questions (ce qui est abordé au chapitre huit).

Le premier chapitre reprend l'essentiel du langage Python. Il est conçu pour servir de référence et permet de rendre le manuel auto-suffisant. Si votre maîtrise de Python le permet, vous pourrez entrer directement dans le vif du sujet avec le chapitre deux et les suivants, mais il est conseillé de vérifier régulièrement les spécificités du langage chaque fois qu'un doute ou une anomalie apparaissent.

Il y a dans ce manuel plus de 150 exercices qui sont tous corrigés, ce qui explique son volume. Les scripts et des compléments sont disponibles sur le site des éditions Ellipses et sont accessibles avec le QR-code ci-joint.

1. Il s'agit du programme qui a pris effet en 2021-2022 pour les classes de première année et en 2022-2023 pour les classes de seconde année.

Table des matières

Partie I • **Premier semestre**

Chapitre 1 • **Programmer avec Python** **13**

1.1 Constantes, identificateurs, variables et affectations 13

1.2 Mots réservés du langage 18

1.3 Types prédéfinis avec Python 19

1.3.1 Types numériques : entiers, flottants, complexes 19

1.3.2 Le type None 20

1.3.3 Le type booléen 21

1.3.4 Les conteneurs 22

1.3.5 Opérations sur les listes, ensembles et dictionnaires 28

1.3.6 Exercices 30

1.4 La programmation et les fonctions avec Python 31

1.4.1 Blocs et indentation 31

1.4.2 Instructions conditionnelles 31

1.4.3 Parcours des objets itérables, boucles for 35

1.4.4 Continue, pass 38

1.4.5 Listes en compréhension 39

1.4.6 Boucle while 40

1.4.7 Break ou pas break ? 44

1.4.8 Procédures et fonctions 45

1.4.9 Compléments : sous-procédures et visibilité des variables 50

1.4.10 Exercices 51

1.5 Modules ou bibliothèques 54

1.5.1 L'instruction import 54

1.5.2 Tableaux (array) de la bibliothèque numpy 56

1.5.3 Les graphiques avec matplotlib 60

1.6 Corrigés des exercices 70

Chapitre 2 • **Quelques algorithmes itératifs fondamentaux** **95**

2.1 Au début était l'arithmétique 96

2.1.1 La division euclidienne des entiers 96

2.1.2 L'écriture binaire d'un entier 98

2.2 Calcul de la moyenne et de la variance 100
2.2.1 Calcul conjoint et analyse 100
2.2.2 Méthode des trapèzes 102
2.3 Algorithmes de recherche séquentielle 103
2.3.1 Listes ou tableaux 103
2.3.2 Recherche d'un élément dans une liste 104
2.3.3 Recherche des plus grands éléments d'une liste 106
2.3.4 Dictionnaires et comptage 110
2.4 Boucles imbriquées 114
2.4.1 Valeurs les plus proches dans un tableau 114
2.4.2 Recherche d'une sous-chaîne dans une chaîne de caractères 115
2.5 Algorithmes dichotomiques 117
2.5.1 Algorithme de recherche dichotomique 117
2.5.2 Exponentiation rapide, version itérative 120
2.6 Corrigés des exercices 122
Chapitre 3 • **Récursivité** **143**
3.1 Introduction 143
3.1.1 Vocabulaire, premiers exemples 143
3.1.2 Quelques dessins de fractales 147
3.2 Mise en garde concernant l'efficacité des programmes récursifs 151
3.3 Mise en oeuvre 157
3.4 Diviser pour régner 160
3.4.1 Recherche de deux points réalisant la plus petite distance 161
3.4.2 Exponentiation rapide, version récursive 163
3.4.3 Recherche dichotomique dans une liste triée 164
3.4.4 Illustration avec des tris 166
3.5 Corrigés des exercices 167
Chapitre 4 • **Les tris** **189**
4.1 Introduction 189
4.2 Tri par insertion 190
4.3 Tri rapide : diviser pour régner 192
4.4 Tri fusion 194
4.5 Tri par insertion dichotomique 196
4.6 Complexité des tris 198
4.7 Sort dans Python 199
4.8 Recherche de la médiane en temps linéaire 200
4.9 Corrigés des exercices 202

Chapitre 5 • **Algorithmes gloutons** **207**
5.1 Introduction 207
5.1.1 Optimisation combinatoire 207
5.1.2 Le principe des algorithmes gloutons 210
5.2 Mise en oeuvre de stratégies gloutonnes 211
5.2.1 Le rendu de monnaie 211
5.2.2 Sélection d'activités 213
5.2.3 Allocations de salles de cours 216
5.3 Bilan 219
5.4 Corrigés des exercices 220

Chapitre 6 • **Traitement de l'image** **233**
6.1 Représentation des images, formats, outils 233
6.1.1 Tableaux à plusieurs dimensions et représentation des images 233
6.1.2 Des fichiers images vers les tableaux de numpy 234
6.2 Dessiner une droite à l'écran 240
6.3 Transformations géométriques d'une image 243
6.3.1 Agrandir : Homothétique d'une image avec k > 1 244
6.3.2 Mode d'emploi pour faire tourner une image 245
6.4 Traitements par convolution 249
6.4.1 Filtres linéaires et convolution 249
6.4.2 Quelques effets du filtrage linéaire 255
6.4.3 Détection de contours 256
6.5 Corrigés des exercices 262

Partie II • **Deuxième semestre**

Chapitre 7 • **Calcul numérique : problématique et outils** **279**
7.1 Représentation des nombres et erreurs de calcul 279
7.1.1 Numérations décimale, binaire, hexadécimale 280
7.1.2 Représentation des entiers sur n bits 282
7.1.3 Les entiers multi-précision de Python 285
7.1.4 Représentation des flottants sur n bits 288
7.1.5 Peut on calculer avec les flottants ? 293
7.1.6 Exercices 299
7.2 Corrigés des exercices 302

Chapitre 8 • **Preuves et complexité des programmes** **311**
8.1 Spécification d'un algorithme 311
8.1.1 Le vocabulaire 311
8.1.2 Vérifier les pré-conditions et les post-conditions 312
8.1.3 Exemples 315
8.2 Le point sur la notion de preuve d'un algorithme 316
8.3 Le point sur la notion de complexité 321
8.3.1 La place, le temps, la précision 321
8.3.2 Les outils : théorie et pratique 323
8.3.3 Exemples basiques 329
8.3.4 Complexité de l'algorithme d'Euclide 330
8.4 Analyse des programmes récursifs 334
8.5 Exercices 337
8.6 Corrigés des exercices 340

Chapitre 9 • **Graphes** **365**
9.1 Définitions 365
9.2 Listes d'adjacence 367
9.3 Matrices d'adjacence 369
9.4 Parcours en profondeur, composantes connexes 370
9.5 Parcours, versions itératives 375
9.5.1 Piles et files 375
9.5.2 Parcours en largeur, procédure itérative 375
9.5.3 Parcours en profondeur, version itérative 378
9.6 Un algorithme de plus court chemin 381
9.7 Graphes, amas et percolation 386
9.8 Corrigés des exercices 390

Chapitre 10 • **Un bref aperçu de la programmation objet** **405**
10.1 Les concepts de la programmation objet 405
10.2 Une classe pour la structure de graphe 406
10.3 Percolation, une implémentation objet 411

Partie III • **Troisième semestre**

Chapitre 11 • **Bases de données, langage SQL** **421**
11.1 Introduction 421

11.2 Qu'est ce qu'une base de données relationnelle ? 422
11.2.1 Les relations comme ensembles de p-uplets 422
11.2.2 Modèle relationnel 423
11.3 Algèbre relationnelle 426
11.3.1 La sélection 427
11.3.2 La projection 427
11.3.3 Le produit cartésien de deux tables, le renommage et la jointure428 428
11.3.4 La jointure 429
11.3.5 Conflits de noms d'attributs et renommage 430
11.3.6 Union, intersection et différence 430
11.3.7 Récapitulatif et expressions de requêtes avec l'algèbre relationnelle 431
11.4 Langage de manipulation de données, SQL 433
11.4.1 Les interrogations 433
11.4.2 GROUP BY, HAVING et les fonctions d'agrégation 439
11.5 Exercices 442
11.6 SQLite, PosgtreSQL, MySQL 445
11.7 Corrigés des exercices 447
Chapitre 12 • **À propos des dictionnaires** **463**
12.1 Dictionnaires ou tableaux associatifs 463
12.1.1 Tableau associatif comme type de données 463
12.1.2 Tableaux associatifs et tables de hachage 466
12.1.3 Quelques fonctions de hachage 467
12.2 Compression de texte : LZ78 469
12.3 Corrigés des exercices 471
12.4 Annexe 474
Chapitre 13 • **Programmation dynamique** **477**
13.1 Premiers exemples 477
13.1.1 Plus longue sous-suite commune 477
13.1.2 Produit de matrices, parenthésages optimaux 482
13.1.3 Problèmes éligibles à la programmation dynamique 486
13.2 Autres exemples 488
13.2.1 Distance d'édition 488
13.2.2 Algorithme de Roy-Floyd-Warshall 490
13.3 Corrigés des exercices 493

Chapitre 14 • **Algorithmes pour l'étude des jeux** **507**
14.1 Jeux sur graphes 507
14.1.1 Exemples de jeux à deux joueurs 507
14.1.2 Représentation par des graphes, vocabulaire 510
14.2 Calcul des attracteurs dans les jeux d'accessibilité 512
14.2.1 Jeu d'accessibilité 512
14.2.2 Attracteurs et pièges 513
14.3 Algorithme du minimax, heuristiques 517
14.3.1 Arbres 517
14.3.2 L'algorithme du minimax 522
14.3.3 L'algorithme du minimax avec heuristiques 524
14.4 Corrigés des exercices 525
14.5 Une classe pour représenter les arbres 535
Chapitre 15 • **Algorithmes pour l'étiquetage et la classification** **537**
15.1 Vocabulaire et définitions 537
15.1.1 Classement et classification automatique 537
15.1.2 Distances, similarités 539
15.1.3 Inertie d'une partition 541
15.1.4 À propos d'intelligence artificielle 543
15.2 Classement supervisé, k-plus proches voisins 544
15.2.1 L'algorithme 544
15.2.2 Les tests 545
15.3 Classification non supervisée, algorithme des k-moyennes 548
15.4 Bibliothèque scikit-learn 552
15.4.1 scikit-learn.datasets 552
15.4.2 k-plus proches voisins avec scikit-learn 553
15.4.3 k-moyennes avec scikit-learn 553
15.4.4 Lexique français anglais (US) 556
15.5 Corrigés des exercices 557
Glossaire de l'informatique générale 569
Bibliographie 581
Index 585

I

Premier semestre

Chapitre 1

Programmer avec Python

Mode de lecture de ce premier chapitre

Ce chapitre regroupe les éléments du langage Python qui permettent la mise en place de l'enseignement de l'informatique de tronc commun. Il est conçu comme un chapitre de référence auquel vous vous référerez quand vous aurez besoin de vérifier ou de chercher des éléments de syntaxe.

Nous ne présenterons qu'une petite partie des primitives disponibles dans la bibliothèque standard et nous travaillerons avec la version 3.8 de Python. Pour en savoir plus, si le besoin s'en fait sentir, vous pourrez utiliser l'aide en ligne avec la fonction **help(...)** ou encore vous référer à la documentation officielle de Python :

https://docs.python.org/fr/3.8/

Python est un langage **interprété** [1], dont le typage est dynamique, qui permet la programmation **fonctionnelle**, la programmation **impérative** et la programmation **orientée objet**. Nous n'aborderons que très succinctement cette dernière (au chapitre 10) qui est par ailleurs abondamment illustrée par les objets natifs de Python.

1.1 Constantes, identificateurs, variables et affectations

Un langage de programmation permet de traiter des constantes, des variables, des expressions. Ce qui suit est illustré avec Python et il y aurait beaucoup de ressemblances avec la plupart des autres langages.

• Une **constante** est un objet de valeur connue non modifiable qui peut être traité par le langage [2]. Par exemple, 12, 12.3, -5.1, 0.001, 'bonjour', **True, False,** [] (liste vide), [1,2], (1,2), {'a' :1, 'b' :1}. Chaque constante admet un **type** (entier, flottant,

1. Les mots en gras non définis dans le texte sont souvent expliqués dans le glossaire. En l'occurrence page 573.

2. Dans certains langages, comme en C, on peut affecter une constante à un identificateur qui ne supportera pas d'autre affectation au long de l'exécution du programme. Cet identificateur est déclaré comme constante.

chaîne de caractères, booléen, liste, tuple, dictionnaire, etc.) qui définit les opérations que cette valeur supporte : algébriques sur les nombres, logiques sur les booléens, accès à élément d'indice donné pour les tuples ou les listes, modification ou insertion d'un élément pour les listes.

• Une **variable** est l'association d'un **identificateur** et d'une valeur (ou constante) stockée en mémoire. Un identificateur est une suite de symboles commençant soit par une lettre soit par le signe _ (underscore) suivi de lettres et/ou de chiffres ou encore de _ qui doit par ailleurs être et différente des mots réservés du langage (dont la liste est en page 18).

• Cette association est réalisée par l'**affectation** d'une valeur à la variable. C'est l'opération qui rend cette dernière utilisable. Au cours de l'exécution d'un programme Python, une même variable peut prendre des valeurs de différents types [3]. La syntaxe d'une affectation est <variable> = <valeur>.

```
>>> x = 7
>>> y = x
>>> z = x+1
>>> x,y,z
(7, 7, 8)
>>> x = y = 12
>>> x, y
(12, 12)
>>> y = 1
>>> x, y
(12, 1)
```

```
>>> x1 = 8
>>> y, z = x1, 2*x1
>>> x, y, z
(12, 8, 16)
>>> 34**3
39304


>>> a=_
>>> a
39304
```

Remarque : La variable "_" joue un rôle particulier : elle contient la dernière expression évaluée (qui est le contenu de l'**accumulateur**). C'est le ans() des calculettes TI ; son usage est réservé aux consoles pour des raisons évidentes de lisibilité et de sécurité du code.

Exercice : On suppose que x et y ont été préalablement affectées. Que donnent les instructions successives `x = y; y = x;` ? Ont-elles permis de permuter les contenus de x et de y ?

3. Dans d'autres langages les variables doivent être préalablement déclarées avec leur type qui est celui des constantes qui leur seront affectées ; ce type, contrairement à ce qui se passe avec Python est invariable. On dit que ce typage est **statique**, dans le cas de Python, on parle de **typage dynamique**.

• **Affectation multiple**
Python autorise l'affectation multiple ce qui est illustré dans la colonne de gauche du tableau qui précède. Nous montrons ci-dessous comment on permute, grâce à cela, le contenu de deux variables. La colonne de droite, quant à elle, montre comment on procède à l'aide d'une variable auxiliaire dans un langage sans affectation multiple. Vous devez savoir le faire !

```
>>> x,y =10,11
>>> x,y
(10, 11)
>>> x,y = y,x
>>> x,y
(11, 10)
```

```
>>> x,y =10,11
>>> z = x
>>> x = y
>>> y = z
>>> x,y
 (11, 10)
```

• **L'incrémentation += et ses variantes**
On peut coder l'incrémentation `n = n+1` de façon plus concise avec `n += 1` et décliner cela avec les opérateurs arithmétiques +, -, * ...

```
i = 0
while i < 10:
    i  +=1
    print(i)

1
2
3
4
5
6
7
8
9
10
```

```
i = 10
while i > 0:
    i  -=1
    print(i)

9
8
7
6
5
4
3
2
1
0
```

```
i = 1
while i < 1030:
    i  *=2
    print(i)

2
4
8
16
32
64
128
256
512
1024
2048
```

• **L'opérateur := (à sauter en première lecture)**
L'opérateur := permet de réaliser une affectation dans une autre instruction comme on le montre avec une instruction conditionnelle (les deux programmes ont le même

effet, dans le second l'affectation est codée dans l'instruction). Il est présent à partir de la version 3.8 de Python et ne devrait pas vous être indispensable.

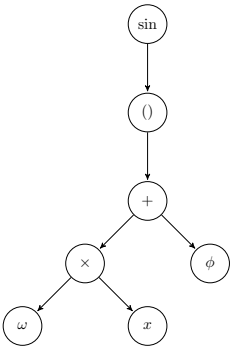
```
L = [1,2,3]
n = len(L)
if n < 10:
    print('L a %s éléments, c\'est trop petit!'%(n))

>>> L a 3 éléments, c'est trop petit!


L = [1,2,3]
if (m := len(L)) < 10:
    print('L a %s éléments, c\'est trop petit!'%(m))

>>> L a 3 éléments, c'est trop petit!
```

• **Une expression** est une combinaison, construite selon les règles de syntaxe du langage, formée de constantes, variables, opérateurs et fonctions. Par exemple :

— si x, y, a, b, c sont des constantes numériques ou des variables qui ont déjà été affectées de valeurs numériques, `a*x**2+b*x*y+c*y**2` qui est l'écriture en Python de $ax^2 + bxy + cy^2$ est une expression numérique valide ;
— la liste [a,b,c,x*y] est elle aussi une expression ;
— si la variable x contient une chaîne de caractères, x + 'autre chaîne' est encore une expression valide (l'opérateur + entre deux chaînes désigne la **concaténation** en Python).

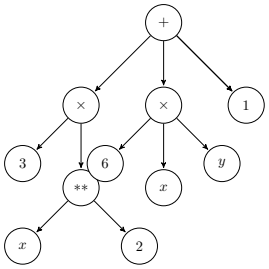
Arbre syntaxique associé à une expression

On définit formellement les expressions comme des arbres finis dont les feuilles sont des opérandes (variables ou constantes), les nœuds non terminaux des opérateurs ou des fonctions. Le nombre de sous-arbres dépend de **l'arité** de l'opérateur que le nœud représente.
Par exemple, $\sin(\omega x + \phi)$ est une expression mathématique valide. L'arbre syntaxique qui lui est associé est représenté à droite.



Exercice 1.1 *représentation des expressions par des arbres*

1. Quelle est l'expression associée à l'arbre qui figure à droite ?
2. On y représente une addition, une multiplication avec 3 branches dérivées, pourtant il s'agit d'opérateurs binaires, pourrait-on faire la même chose avec l'élévation à la puissance ?



3. Réécrire cette même expression avec un arbre binaire (dont les nœuds ont au plus deux fils). *Corrigé en 1.1, page 70*

Séparateurs...

— le **point-virgule** sépare deux **instructions** sur une même ligne dans le shell ou dans un script ;
— la **virgule** entre deux **expressions** à l'intérieur de crochets [] est un constructeur de **liste** ;
— la **virgule** entre deux **expressions** à l'intérieur de () ou pas, est un constructeur de **tuple** ;
— les **espaces** et l'**indentation** tiennent lieu de délimiteurs syntaxiques ; nous détaillons cela avec l'apprentissage de la programmation, page 31 ;
— le **double point** n'est pas un séparateur d'instructions, c'est un composant syntaxique des boucles et instructions conditionnelles.

```
>>> a = 123;  b = 145
>>> a,b
(123, 145)
>>> a; b
123
145
>>> z = a; a = b; b = z
>>> a,b
(145, 123)
>>>  c = 12
SyntaxError: unexpected indent
>>> c = 12
>>> for i in range(0,3):
        print(i);
>>> L =[a,]; L
 [145, 123]
```

Illustration dans une console :
- observez la différence entre a ; b et a,b
- soyez attentifs au message d'erreur lorsqu'il y a un espace en début de ligne dans `>>>  c = 12`

1.2 Mots réservés du langage

Ce sont les mots du langage standard ; ils ne peuvent servir d'identifiants. Le tableau indique leur contexte d'utilisation et la page où ils sont présentés dans ce cours lorsque c'est le cas.

mot	contexte	page
and	connecteur logique binaire (expressions booléennes)	21
as	associée à **import** dans '**import** package **as** ...'	55
assert	après évaluation d'une expression booléenne, permet de lever l'exception AssertionError	313
async	non traité ici	
await	non traité ici	
break	provoque l'abandon d'une boucle **for** ou **while**	44
class	déclaration d'une classe (programmation orientée objet)	405
continue	dans une boucle, permet de sauter l'étape en cours	38
def	déclaration d'une fonction	45
del	permet d'effacer un ou plusieurs éléments à partir de leurs indices	28, 29
elif	dans une instruction conditionnelle **if...elif...else**	31
else	dans une instruction conditionnelle **if...elif...else**	31
except	associé à try pour la gestion des erreurs	...
False	une des deux constantes booléennes	21
finally	associé à try pour la gestion des erreurs	...
for	définit une boucle **for ... in...**	35
from	'**from** package **import** ...'	54
global	déclaration des variables globales dans une fonction	47
if	définit une instruction conditionnelle **if...elif...else**	31
import	dans '**import** package '	54
in	associé à **for** pour définir une boucle	21
in	opérateur booléen ; relation d'appartenance à un conteneur	21
is	teste l'égalité de deux objets (comme ==)	21
lambda	définition d'une fonction par une expression	49
None	voir l'usage avec les définitions de fonctions et procédure	45
nonlocal	gestion de la visibilité d'une variable dans une sous-procédure (à associer à local et global)	50
not	opérateur logique unaire	21
or	connecteur logique binaire (expressions booléennes)	21
pass	instruction vide (pratique en cours de programmation)	38
raise	levée d'une exception	...

return	signale la valeur que retourne une fonction	45
True	une des deux constantes booléennes	21
try	instruction permettant de gérer (capturer) des erreurs ou exceptions	...
while	définit une boucle conditionnelle	40
with	simplification d'écriture ; associé à **as** (non traité ici)	...
yield	associé à la notion de co-routine (non traité ici)	...

1.3 Types prédéfinis avec Python

1.3.1 Types numériques : entiers, flottants, complexes

• Les entiers

On représente les entiers (éléments de $\mathbb{Z}$) par des objets de type **int** (pour integer ou entier). Les opérations arithmétiques usuelles sont définies comme sur une calculette : +, - (relation binaire, soustraction), - (unaire, changement de signe), $\star$ (multiplication), $\star\star$ (élévation à la puissance) ; a**//**b désigne le quotient dans la **division euclidienne** de a par b, le reste est `a%b`, **divmod**(a,b) est le couple (q, r) où la relation $a = bq + r, 0 \leq r < b$ définit (q, r) de façon unique.

• Les flottants

On approche les réels par des objets de type **float** (pour float ou flottant).
Les constantes de type float ont un affichage décimal (comme 12.3) ou scientifique (comme 2.4379168015552228e-36). Les opérations usuelles sont encore définies : +,- (unaire et binaire), *, /, ** (avec $a * *b = e^{b \ln a}$ si b n'est pas entier) ; comme les opérations sur les entiers, elles obéissent aux mêmes **règles de priorité** que celles de vos calculettes et qui sont nos règles de calcul et de parenthésage habituelles. Les conversions de bon sens [4] pour les expressions mêlant entiers et flottants sont assurées (a+x avec a entier et x flottant retourne un flottant), la division de deux entiers a/b retourne le quotient approché (on la distinguera donc de l'expression retournant le quotient dans la division euclidienne a//b).

```
>>>a=36789; b=563
>>> a//b
65
>>> divmod(a,b)
(65, 194)
>>> a/b
65.34458259325045
```

```
>>> type(a/b)
<class 'float'>
>>> type(a//b)
<class 'int'>
```

4. Il s'agit de votre bon sens, pas de celui de la machine.

• Les complexes
Les complexes sont représentés par des couples de deux flottants à l'aide du constructeur **complex** avec la syntaxe complex(x,y) (x et y flottants) ou encore avec une constante de la forme 1+1j. Les opérations usuelles sur les complexes sont évidemment **implémentées** : +,-,*,/,**, parties réelle, imaginaire, conjugué, module...

Le complexe i est noté **1j**, on définira $x + iy$ avec **w=complex(x,y)** ou **w=x+y*1j**.

Construction des complexes et opérations :
- on illustre les deux façons de construire un complexe ;
- on prendra garde aux différentes syntaxes des opérations : **abs(z)** comme une fonction, **z.conjugate()** comme une méthode, **z.real**, **z.imag** comme des champs (ou attributs) de classe. *Ce qui pour le moment paraît être un total désordre prendra tout son sens lorsque nous parlerons de programmation orientée objet.*
- Le module numpy propose des fonctions numpy.real, numpy.imag, numpy.absolute, numpy.conjugate vectorialisables (nous expliquons cela en section (1.5.2)).

```
>>> z = complex(1,1); z
(1+1j)
>>> w = 1j; w
1j
>>> 1j
1j
>>> 1*j
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    1*j
NameError: name 'j' is not defined
>>> z.real
1.0
>>> z.conjugate()
(1-1j)
>>> z*z.conjugate()
(2+0j)
>>> abs(z)
1.4142135623730951
```

1.3.2 Le type None

Python reconnaît un type et un objet **None.** Nous verrons cela page 45 avec la présentation des fonctions.

1.3.3 Le type booléen

Ce type permet d'effectuer les tests. Il comprend deux constantes **True** et **False** , les expressions booléennes sont donc les expressions logiques. Python transforme tout un tas d'objets en booléens : **nous éviterons d'en user.** On construit des expressions booléennes avec les **opérateurs de comparaison** ==, <, <=, !=, les **connecteurs logiques and, or, not** (négation), **is** (égalité), **in** (appartenance à un conteneur)...

```
>>> A="Bonjour";B='Bonjour'
>>> A is B
True
>>> A==B
True
>>> A!=B
False
>>> 'on' in A
True
>>> 'i' not in A
True
```

```
>>>a=1234; b=5678
>>> a<b
True
>>> a<b or b<a
True
>>> a+b>b
True
>>> x=0; x!=0 and a/x>1
# seule la première clause
est évaluée
False
```

opérateur	évaluation
(P **and** Q)	P et Q sont des expressions booléennes ; si P est évaluée à **True** (ou vraie), Q est alors évaluée. Si Q est vraie, (P **and** Q) est vraie, fausse sinon ; si P est évaluée à **False** (ou fausse), Q n'est pas évaluée et (P **and** Q) est fausse.
(P **or** Q)	P et Q sont des expressions booléennes ; si P est évaluée à **True** (ou vraie), Q n'est pas évaluée et (P **or** Q) est vraie. si P est évaluée à **False** (ou fausse), Q est évaluée et (P **or** Q) est vraie ssi Q est vraie.
not P	P est une expression booléenne ; négation de P
$(E_1 == E_2)$ $(E_1$ **is** $E_2)$	teste l'égalité lorsque E_1 et E_2 sont des expressions (numériques, booléennes ou autres)
$(E_1 != E_2)$ $(E_1$**is not**$E_2)$	retourne la valeur inverse de $(E_1$ **is** $E_2)$ E_1 et E_2 sont des expressions (numériques, booléennes ou autres)

$(E_1 < E_2)$ $(E_1 <= E_2)$ $(E_1 > E_2)$ $(E_1 >= E_2)$	comparaisons (lève une erreur de type si les objets ne peuvent être convertis en objets comparables)
(e **in** S)	teste l'appartenance de e à un conteneur ; retourne une erreur de type si S n'est pas un conteneur.

1.3.4 Les conteneurs

On appelle conteneur des objets qui sont susceptibles d'en contenir d'autres. Ce sont, dans le noyau du langage, les chaînes de caractères, les tuples, les listes, les ensembles et les dictionnaires. Ils acceptent tous la syntaxe <elt> **in** <conteneur> qui renvoie une expression booléenne et teste l'appartenance de <elt> au conteneur (caractère dans une liste, élément dans les tuples, listes et ensembles, clé dans un dictionnaire).
Parmi eux, sont **indexables** les chaînes, tuples et listes, c'est-à-dire que l'on accède à leurs éléments par leur indice avec la syntaxe X[i]. Cette notation n'a pas de sens pour un ensemble dont les éléments ne sont pas ordonnées et pour un dictionnaire la notation D[<clé>] a un sens différent.

Les termes des objets indexables sont numérotés de 0 à longueur - 1

Les chaînes de caractères (string)

Une chaîne de caractères est une suite de caractères quelconques délimités par deux ' (apostrophes) ou deux " (guillemets) indifféremment. La chaîne vide est `''` (ne pas confondre avec un espace `' '`).

fonction, opérateur	effet
+	concaténation de deux chaînes de caractères
float()	convertit un nombre ou une chaîne en flottant
int()	convertit un nombre ou une chaîne en entier
str()	convertit un nombre (ou un objet quelconque pour l'affichage par exemple) en une chaîne
ch.count(s)	nombre de sous-chaînes égales à s dans ch
ch.replace(s1,s2)	renvoie une autre chaîne construite en remplaçant la sous-chaîne s1 par s2 dans ch
ch.split( )	renvoie la liste des mots de ch séparés par un espace ' ' (par défaut)
ch.strip()	renvoie une autre chaîne construite en supprimant les espaces (et symboles de fin de ligne et tabulations) en début et fin de chaîne
s.join(L)	L : liste de chaînes, s : chaîne (str), renvoie la chaîne L[0]sL[1]s...sL[-1]

On peut déterminer la **longueur** d'une chaîne X (len(X)), en **extraire** des caractères (le premier caractère étant X[0], le dernier X[len(X)-1]) ; on peut **concaténer** deux chaînes ou plus, convertir une chaîne formée de chiffres en nombre... Un tel objet est de type **str** (pour string ou chaîne).

Chaînes de caractères : opérations et fonctions de base : concaténation, comparaison (is), longueur, indexage

```
>>> A='Bonjour'
>>> type(A)
<class 'str'>
>>> B="Bonjour"
>>> A is B
True
>>> A+" "+B
'Bonjour Bonjour'
>>> len(A)
7
>>> A[0]
'B'
>>> A[1]
'o'
>>> A[0],A[1],A[2]
('B', 'o', 'n')
```

Accès par tranches attention, on peut écrire ch[début : len(ch)], mais pas ch[len(ch)]

```
>>> A[0:len(A)]
'Bonjour'
>>> A[7]
Traceback (...):
 File "<pyshell#56>",line
 1,  in <module>  A[7]
 IndexError: string index
 out of range
>>> A[6]
'r'
>>> int('2')
2
>>> int('23')+7
30
>>> float('23.1')
23.1
```

La conversion est par exemple indispensable lorsqu'on propose à l'utilisateur d'un programme d'entrer des nombres au clavier (ils sont lus comme des chaînes de caractères) :

```
x =int(input('x = '))
print(x+2)

x = 2
4
```

```
 x =input('x = ')
 print(x+2)
>>>
x = 2
Traceback  bla bla bla
TypeError: Can't convert 'int' object to str implicitly
```

Tuples et listes

Tuples
Regardez bien la dernière ligne : le découpage en tranches (slicing) **objetindexable[debut :fin]** retourne dans tous les cas les termes de <objet indexable> d'indices **début à fin -1** .

```
>>> t=(12); type(t)
<class 'int'>
>>> t=(12,); type(t)
<class 'tuple'>

>>> t=(1,(2,3),4,(5,6,7),8,
                        'fin')
>>> t
(1, (2, 3), 4, (5, 6, 7), 8,
                        'fin')
>>> t[0]
1
>>> t[1]
(2, 3)
>>> len(t)
6
>>> t[6]
Traceback ...bla bla bla
IndexError: tuple index out of
range
```

Listes
On prendra garde à la différence entre la concaténation (+) qui produit une autre liste (ailleurs en mémoire) et les méthodes **insert**, **append**, **pop**... qui modifient L (dont l'adresse mémoire est inchangée).

```
>>> L=[1,2,4,4]
>>> L.append(0)
>>> L
[1, 2, 4, 4, 0]
>>> L.insert(0,777); L
[777, 1, 2, 4, 4, 0]
>>> 7 in L
False
>>> 777 in L
True
>>> L+[567]
[777, 1, 2, 4, 4, 0, 567]
>>> L
[777, 1, 2, 4, 4, 0]
>>> L.append(567)
>>> L
[777, 1, 2, 4, 4, 0, 567]
>>> L.pop(4);L
4
[777, 1, 2, 4, 0, 567]
```