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.

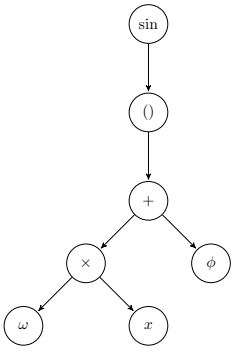
```
L = [1,2,3]
n = len(L)
if n < 10:
    print('L a %s éléments, c\'est trop petit!'%(n))

>>> 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).

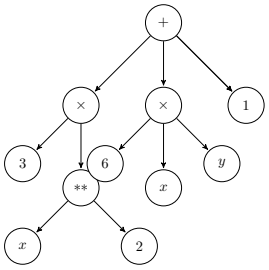
Arbre syntaxique associé à une expression

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.