Chapitre 1

Programmer avec Python

Mode de lecture de ce premier chapitre

Ce chapitre est dévolu à l'apprentissage du langage de programmation avec lequel nous mettrons en place les notions de base de l'algorithmique. Il commence par la présentation des différents types des objets que nous aurons à manipuler. Ce n'est pas la façon la plus ludique de procéder, mais nous partons du point de vue que vous avez déjà écrit des micro-programmes au lycée et notre objectif est un apprentissage efficace, donc structuré. Vous pourrez d'ailleurs faire des aller-retours entre la partie (1.3) et ce tout début de chapitre. Il est conseillé de reproduire les exemples encadrés, de jouer avec le logiciel pour tester ses idées et vérifier la cohérence des règles énoncées. Des exercices de premier niveau sont là, dès le début, pour faciliter la mémorisation, mettre en lumière les subtilités ou les particularités de la syntaxe de Python. Avec la section (1.3) on aborde le cœur du sujet et les exercices visent alors à conforter l'apprentissage des techniques, incitent aux calculs de coûts et à la justification des algorithmes.

Nous ne présenterons qu'une petite partie des primitives disponibles dans la bibliothèque standard et nous travaillerons avec la version 3.2 de Python (la dernière pour laquelle sont écrits les modules numpy et scipy au moment où nous l'écrivons). Pour en savoir plus, si le besoin s'en fait sentir, on pourra utiliser l'aide en ligne, la fonction **help(...)** ou encore se référer au site

http ://docs.python.org/3.2/library/functions.html

1.1 Constantes, identificateurs, variables, affectation ...

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

• Une **constante** est un objet qui désigne une valeur connue non modifiable, par exemple, 12, 12.3, -5.1, 0.001, 'bonjour', **True, False .** Chaque constante admet un **type** (entier, flottant, booléen, chaîne de caractères, liste, tuple etc...) qui définit les opérations qu'elle supporte : algébriques sur les nombres, logiques sur les booléens, accès à un emplacement 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 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 _ . Elle doit par ailleurs être et différente des **mots réservés** du langage.

• Cette association est réalisée par l'**affectation** d'une valeur à la variable. C'est l'opération qui rend cette dernière utilisable [1]. La syntaxe d'une affectation est
<variable> = <valeur>.
On peut savoir que 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, de Scilab, le % de Maple etc... ; son usage est réservé au shell pour des raisons évidentes de lisibilité et de sécurité du code.

```
>>> x=7
>>> y,z=x,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
```

Exercice : que donnent les instructions successives `x = y; y = x;` ?

• 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 !

1. 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** ; pour Python on parle de **typage dynamique**

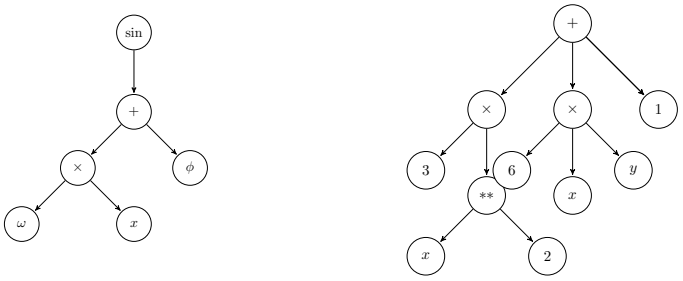
```
>>> x,y =10,11
>>> x,y
(10, 11)
>>> x,y = y,x
>>> x,y
(11, 10)
```

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

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

– si x, y, a, b, c sont des constantes numériques ou des variables qui ont déjà été affectées de valeurs numériques, $ax^2 + bxy + cy^2$ est une expression numérique valide, que l'on écrira `a*x**2+b*x*y+c*y**2` ;
– 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).

⋆ 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. Par exemple, $\sin(\omega x + phi)$ est une expression valide dès que les variables x et phi ont été affectées. L'arbre syntaxique qui lui est associé est représenté à gauche.



Exercice :

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'élevation à la puissance ?
3. Réécrire cette même expression avec un arbre dont les nœuds ont au plus deux fils.

Mots réservés Ce sont les mots du langage standard ; ils ne peuvent servir de variables. Le tableau indique leur contexte d'utilisation et la page où ils sont présentés dans ce cours.

mot	contexte	page
and	connecteur logique binaire (expressions booléennes)	26
as	associée à **import** dans '**import** package **as** ...'	37
assert	après évaluation d'une expression booléenne, permet de lever l'exception AssertionError (non traité ici)	...
break	provoque l'abandon d'un boucle **for** ou **while**	54
class	voir le chapitre consacré à la programmation objet	403
continue	dans une boucle, permet de sauter l'étape en cours	48
def	permet la définition d'une fonction	55
del	permet d'effacer un ou plusieurs éléments à partir de leur index	31
elif	dans une instruction conditionnelle **if**	41
else	dans une instruction conditionnelle **if**	41
except	associé à try pour la gestion des erreurs	169
False	une des deux constantes booléennes	25
finally	associé à try pour la gestion des erreurs	169
for	définit une boucle **for ... in...**	45
from	'**from** package **import** ...'	34
global	déclaration des variables globales dans une fonction	57
if	définit une instruction conditionnelle **if**	41
import	dans '**import** package '	34
in	associé à **for** pour définir une boucle	45
in	opérateur booléen ; relation d'appartenance à un conteneur	45
is	teste l'égalité de deux objets (comme ==)	26
lambda	définition d'une fonction par une expression	58
None	voir l'usage avec les définitions de fonctions et procédure	55
nonlocal	gestion de la visibilité d'une variable dans une sous-procédure (à associer à local et global)	59
not	opérateur logique unaire	26
or	connecteur logique binaire (expressions booléennes)	26
pass	instruction vide (pratique en cours de programmation)	48
raise	levée d'une exception	170
return	signale la valeur que retourne une fonction	55
True	une des deux constantes booléennes	25
try	instruction permettant de gérer (capturer) des erreurs ou exceptions	169
while	définit une boucle conditionnelle	50
with	simplification d'écriture ; associé à **as** (non traité ici)	...
yield	associé à la notion de co-routine (difficile)	...

Séparateurs...

– le **point-virgule** sépare deux **instructions** sur une même ligne dans le shell ou dans un script ;
– la **virgule** entre deux **expressions** est un constructeur de séquence ;
– le double point n'est pas un séparateur d'instructions, c'est un composant syntaxique des boucles et instructions conditionnelles ;
– les **espaces** et l'**indentation** tiennent lieu de délimiteurs syntaxiques ; nous détaillons cela avec l'apprentissage de la programmation, page 40.

Illustrations dans le shell - 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	**Illustrations dans l'éditeur de scripts (pour lecteur ayant déjà programmé)** - on pourrait aussi terminer les lignes par un ; - on prendra garde à l'indentation : le moindre décalage et boum !

`>>> 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);`
`0`
`1`
`2`
	`n=120`
	`p = 1`
	`for k in range(1,n+1):`
	`  p=p*k`
	`print(p, sep="...")`
	
	`def myfact(n):`
	`  p=1`
	`  for k in range(1,n+1):`
	`      p=p*k`
	`  return p`

1.2 Types prédéfinis avec Python

1.2.1 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 [2] 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'>
```

• 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**

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

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 (1.2.8))

```
>>> 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
```

1.2.2 Le type None

Python reconnaît un type et un objet **None.** Nous verrons cela page 55 avec la présentation des fonctions.

1.2.3 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.