Introduction à l'algorithmique et à la programmation avec Python

Laurent Signac
https://deptinfo-ensip.univ-poitiers.fr

27 septembre 2017

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those !

```
>>> import this
```

Table des matières

Notations

Dans ce support de cours, les programmes sont généralement typographiés en police machine à écrire et encadrés, alors que les algorithmes n'ont pas de cadre et utilisent une police sans empattements.
Sauf si une indication contraire est donnée, le langage utilisé est Python. On différencie les «sessions shell» par la présence du prompt `>>>`. En l'absence de prompt, il s'agit généralement d'un programme à entrer dans un éditeur de texte et à exécuter ensuite (l'interface IDLE convient très bien pour faire tout ceci).

Introduction

L'informatique, telle que nous la connaissons aujourd'hui résulte à la fois des progrès théoriques, amorcés dans les années 30 par les travaux de Turing, et des progrès technologiques, liés à l'électronique, avec en particulier l'invention du transistor dans les années 40.
Depuis, ces deux facettes, science théorique et défis technologiques progressent de conserve.
L'objet de ce cours est de présenter l'ère numérique sous l'angle du codage des informations, puis celle des ordinateurs sous celui des algorithmes et de la programmation.

Ressources

La dernière version de ce fichier est disponible ici : https://deptinfo-ensip.univ-poitiers.fr/FILES/PDF/python1a.pdf
Les ressources Web associées à ce cours sont :

- Sur Updago : Algorithmique et Programmation Python
- https://deptinfo-ensip.univ-poitiers.fr/ENS/doku

Les introduction à Python sont nombreuses sur le Web, et certaines sont de très bonne qualité :

- Le cours de Bob Cordeau :
  http://perso.limsi.fr/pointal/python:courspython3
- Le cours de Pierre Puiseux :
  http://web.univ-pau.fr/~puiseux/enseignement/python/python.pdf

Voici ceux qui ont été utilisés lors de l'écriture de ce cours [1] :

- LE GOFF, Vincent, *Apprenez à Programmer en Python*, Le livre du Zéro
- PUISEUX, Pierre, *Le Langage Python*, Ellipses TechnoSup
- CHAZALLET, Sébastien, *Python 3 – Les fondamentaux du langage*, Eni
- ZELLE, John, *Python Programming*, Franklin, Beedle, and Associates
- LEE, Kent D., *Python Programming Fundamentals*, Springer
- SUMMERFIELD, Mark, *Programming in Python 3*, Addison Wesley, 2e ed.

Les parties du cours qui ne concernent pas précisément Python sont issues de nombreux ouvrages impossibles à tous mentionner ici et de quelques années de pratique.

Licence

1. Depuis, l'offre de livres sur Python 3 s'est diversifiée.

Chapitre I

Ordinateur, codage numérique

Il y a 10 sortes de gens, ceux qui savent compter en binaire et les autres.

1 Ordinateur ?

Les machines électroniques nous entourent désormais : de la simple calculatrice aux super-ordinateurs, en passant par les *smartphones*, les tablettes et les micro-ordinateurs. À partir de quel degré de complexité avons-nous affaire à un *ordinateur* ?
Comme nous allons le voir, les ordinateurs actuels sont à peu près calqués sur un modèle datant des années 50, le modèle de Von Neumann. Si nous devions retenir deux caractéristiques des ordinateurs, ce serait que :

- un ordinateur doit être programmable
- son programme doit être enregistré dans sa mémoire.

Par voie de conséquence, une simple calculatrice de poche n'est pas vraiment un ordinateur, alors que les modèles programmables plus perfectionnés en sont. De même que les tablettes et les *smartphones*...

1.1 Ordinateurs domestiques

Dans nos micro-ordinateurs, les différents composants sont : la carte mère, le processeur, la mémoire, la carte graphique, les disques...
Ces composants électroniques communiquent entre eux par l'intermédiaire de *bus* et les données qui transitent sont représentées sous forme *binaire*.
Dans un ordinateur, absolument tout (données et programmes, en mémoire, sur les disques durs ou sur CD...) est stocké sous forme binaire.
Dans le suite, nous verrons comment utiliser le système binaire, et comment des objets complexes, comme des images ou de la musique, sont codées en binaire.

Obtenir plus d'informations

Le film suivant :
http://www-sop.inria.fr/science-participative/film/
a été produit par l'INRIA et retrace l'histoire de l'informatique du 9e siècle à aujourd'hui en 24 minutes...

2 Codage numérique de l'information

L'information est la matière première de l'informatique[1]. Les algorithmes, qui sont constitués d'informations, stockent, manipulent et transforment d'autres informations, à l'aide de machines.
Tout, en informatique, est représenté *comme une séquence de 0 et de 1* : les algorithmes, ou plutôt les programmes, le contenu d'un livre, une photo, une vidéo...

1. Le mot vient d'ailleurs de là : **informa**tion auto**matique**, et a été adopté en 1967.

Pourquoi le binaire ?

Il y a une raison théorique et une raison technique à l'utilisation du binaire :

- on ne peut pas faire plus simple que 2 symboles. Avec un seul symbole, plus rien ne fonctionne (on a un seul codage de longueur fixe, la taille de l'écriture d'un nombre, écrit en unaire est proportionnelle au nombre, et non pas à son logarithme).
- les deux symboles 0 et 1 sont transposables électroniquement par : le courant passe ou ne passe pas (système assez robuste)

On représente donc toutes les informations sous forme de *bits* (binary digit = chiffre binaire). La quantité d'information est justement le nombre de bits nécessaires pour représenter cette information.

Unités

- le *bit* (*binary digit*) est l'unité d'information : 0 ou 1
- l'*octet* est un groupe de 8 bits (attention, octet se dit *byte* en anglais)

Tout le reste.... dépend du contexte. En particulier, le préfixe «kilo» correspond dans le système international au multiplicateur 10^3, mais conventionnellement, il était utilisé en informatique pour le multiplicateur 2^{10}. La vérité [2] est que, maintenant, nous devrions utiliser les préfixes du SI pour le multiplicateur 10^3 (symboles k,M,G...) et d'autres préfixes (kibi, mébi, gibi,... symboles ki,Mi,Gi...) pour les multiplicateurs 2^{10}, 2^{20}, 2^{30}...Il en est de même pour les débits (bits/seconde, puis kbits par seconde etc...). Dans les symboles, le «o» de octet est parfois remplacé par le «B» de byte, à ne pas confondre avec le «b» de bit...

Enfin, l'explorateur Windows utilise le symbole Ko pour 2^{10} octets (alors que ce devrait être 10^3 octets) et l'utilitaire GNU `du -k` utilise aussi des kilos de 1024 octets (encore qu'il a une option supplémentaire permettant d'utiliser les kilos du SI).

Dans le doute, partez du principe que si on vous vend de la mémoire (disque dur, mémoire flash...) sa taille est indiqué en utilisant le SI...

2.1 Changements de base

L'écriture d'un nombre en base b est formée à partir de b symboles, appelés chiffres. Les 10 chiffres de la base 10 sont par exemple : $0, 1, 2, 3, 4, 5, 6, 7, 8$ et 9.
Si l'écriture d'un nombre en base b est : $c_n c_{n-1} ... c_1 c_0$ (les c_i sont les *chiffres*), sa valeur sera :

$$v(c_n) \times b^n + v(c_{n-1}) \times b^{n-1} + ... + v(c_1) \times b^1 + v(c_0) \times b^0$$

où $v(c_i)$ est la *valeur* associée au chiffre (symbole) c_i. La valeur du symbole 9 en base 10 est par exemple neuf. Généralement, pour une base b inférieure à 10, on reprend les même chiffres qu'en base 10, avec les mêmes valeurs (mais on n'utilise que les b premiers chiffres). Pour une base b supérieure à 10, on ajoute aux chiffres ordinaires d'autres symboles, comme des lettres. La base 16 utilise par exemple les 16 chiffres suivantes : $0, 1, 2, ..., 9, A, B, C, D, E, F$.
Voici quelques exemple, la base est indiquée en indice à la fin du nombre. En l'absence de cet indication, c'est la base 10 qui est utilisée.

$$1011010|_2 = 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 90$$

$$5A|_{16} = 5 \times 16^1 + 10 \times 16^0 = 90$$

Pour passer de la base 10 à la base 2, on peut procéder par divisions successives (par 2). Voici comment trouver l'écriture binaire de 90 :

$$\begin{array}{rll} 90 & = 2 \times 45 & +0 \\ 45 & = 2 \times 22 & +1 \\ 22 & = 2 \times 11 & +0 \\ 11 & = 2 \times 5 & +1 \\ 5 & = 2 \times 2 & +1 \\ 2 & = 2 \times 1 & +0 \\ 1 & = 2 \times 0 & +1 \end{array}$$

La séquence des restes successifs, lue du dernier au premier donne : 1011010 qui est l'écriture en base 2 de 90.
On peut procéder de même en base 16 (en faisant des divisions par 16). Les restes sont alors entre 0 et 15, ce qui correspond bien aux valeurs des chiffres de la base 16.
Les nombres non-entiers sont codés de la même manière. Chaque chiffre placé après la virgule est associé à une puissance négative de la base :

$$101,011|_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 5 + 0,25 + 0,125 = 5,375$$

Le passage de la base 10 à la base 2, pour la partie fractionnaire, est réalisé par multiplications successives (on reporte la partie fractionnaire de chaque résultat sur la ligne suivante, et les parties entières des résultats obtenus forment l'écriture en base 2). Voici comment écrire $0,6875$ en base 2 :

$$\begin{array}{rll} 0,6875 & \times 2 = & 1,375 \\ 0,375 & \times 2 = & 0,75 \\ 0,75 & \times 2 = & 1,5 \\ 0.5 & \times 2 = & 1 \end{array}$$

Nous prenons les parties entières des résultats dans l'ordre où elles sont obtenues : 1011. Donc $0,6875 = 0,1011|_2$.

Développement binaires infinis

Notons que si l'écriture d'un nombre non entier en base 2 est finie, ce sera aussi le cas en base 10 (car les puissances négatives de 2 ont toutes une écriture décimale finie). En revanche, si l'écriture décimale est finie, l'écriture en base 2 ne le sera pas forcément (elle sera alors périodique). Par exemple :

$$\begin{array}{rll} 0,2 & \times 2 = & 0,4 \\ 0,4 & \times 2 = & 0,8 \\ 0,8 & \times 2 = & 1,6 \\ 0,6 & \times 2 = & 1,2 \\ 0,2 & \times 2 = & ... \end{array}$$

En conséquence : $0,2 = 0,0011\overline{0011}...$

2.2 Codage des nombres

Nombres négatifs

Les nombres négatifs sont représentés en binaire en utilisant la méthode du complément à deux, qui permet de réaliser simplement des opérations arithmétiques directement sur le codage.
Pour représenter un nombre avec la méthode du complément à 2, on choisit auparavant le nombre de bits maximum qui sera occupé par le code. Les valeurs typiques sont multiples de 8. Dans la suite, nous utiliserons des compléments à 2 sur 8 bits.
Coder les nombres en complément à 2 sur 8 bits revient à partager les 256 codes possibles en deux parties. Ceux dont le bit de poids fort est 0 représenteront les nombres positifs de 0 à 127 (codage immédiat en base 2 avec les 7 bits restants). Ceux dont le bit de poids fort est 1 représenteront les nombres négatifs de -128 à -1.
Dans le cas d'un nombre négatif n, on obtient le complément à 2 de n en calculant la représentation binaire du nombre $256 - |n|$. Cette représentation est obtenue rapidement en écrivant sur 8 bits la représentation binaire de la valeur absolue de n, puis en inversant les 0 et les 1, puis en ajoutant 1 :

Exemple de codage en complément à deux

Nous souhaitons coder le nombre -66 en complément à deux sur 8 bits. On commence par écrire sa valeur absolue en binaire, sur 8 bits :

01000010

Puis on échange les 0 et les 1 :

10111101

Puis on ajoute 1 (attention aux éventuelles retenues) :

10111110

Le résultat est le codage en complément à 2 sur 8 bits de -66. On remarque que c'est aussi l'écriture binaire de $256 - 66 = 190$.

Virgule fixe

Coder un nombre en virgule fixe consiste à réserver un certain nombre de chiffres pour la partie entière et un certain nombre de chiffres pour la partie décimale. Ainsi, en codant en virgule fixe sur 2 octetes avec un octet pour la partie entière, on peut représenter les nombres de 0 à $256 - 2^{-8}$ avec un «pas» de 2^{-8}. Le fait que le pas soit fixe est un défaut de ce codage.

Virgule flottante

Nous avons vu qu'il était possible de représenter les nombres à virgule en binaire et de les coder par exemple en virgule fixe. Cependant, l'utilisation très fréquente des calculs à l'aide de nombres non entiers, le soucis de coder ces nombres en un minimum de bits et la nécessité d'avoir une erreur «relative» peu importante (pas de pas fixe) a donné naissance au codage en *virgule flottante*. Ce codage (norme IEEE-754) existe essentiellement en deux versions, simple et double précision, utilisant des codages sur 32 et 64 bits. Le codage en simple précision permet de représenter *de manière approchée* les nombres entre -3.4×10^{38} et 3.4×10^{38}, tout en atteignant des valeurs aussi petites (en valeur absolue) que 1.4×10^{-45}.
Nous ne détaillerons pas ici ce codage, assez technique, mais il est important de retenir :

— que c'est le codage le plus utilisé pour les nombres non entiers ;
— qu'il représente les nombres de manière approchée, et qu'en virgule flottante, les calculs ne sont (pratiquement) jamais exacts.

Flops

Une indication de vitesse des processeurs donnée en FLOPS (ou un de ses multiples) fait référence au nombre d'opérations en virgule flottantes par seconde (FLoating point Operations Per Second).

L'ordinateur le plus rapide du monde (depuis 2013[3], c'est Tianhe II à 3 millions de cœurs) a une vitesse estimée de 30 PetaFlops (10^{15}) soit plus de 30 millions de milliards d'opérations en virgule flottante par seconde. Un ordinateur domestique atteint des vitesses de quelques dizaines de GigaFlops (10^9 Flops)

2.3 Numérique vs analogique

Beaucoup d'objets quotidiens existent à la fois en version numérique et analogique, même si la première tend à remplacer petit à petit la seconde.
Ainsi, nous pouvons opposer les disques compacts audio et les disques vinyles, la photo numérique et la photo argentique, les e-book et les livres traditionnels. Dans tous ces cas, l'objet numérique n'est qu'une succession de chiffres 0 et 1...

Codage des couleurs

— Les couleurs, sur un écran, sont formées par synthèse additive (addition de la lumière). L'impression, au contraire, est réalisée par synthèse soustractive.
— Les couleurs primaires en synthèse additive sont : le rouge, le vert et le bleu (en synthèse soustractive, il s'agit du cyan, du magenta, et du jaune).
— Supposons qu'on dispose de 3 leds, une rouge, une verte et une bleue, et que chacune puisse être allumée ou éteinte (on ne peut pas varier leur intensité).

À partir de ces 3 leds, nous pouvons obtenir $2^3 = 8$ couleurs puisque chaque led peut être soit allumée, soit éteinte. Nous obtenons ainsi du noir si toutes les leds sont éteintes, du blanc si elles sont toutes allumées ou du cyan si seules les leds bleues et vertes sont allumées.

Puisqu'à chacune des 8 couleurs obtenues, correspond un état de chacune des trois leds, nous pouvons convenir de coder l'état des leds dans l'ordre rouge, vert, bleu par 0 (led éteinte) ou 1 (led allumée). Ainsi, une série de 3 chiffres binaires, (autrement dit un entier entre 0 et 7) est associé à une couleur :

0	000	noir	4	100	rouge
1	001	bleu	5	101	magenta
2	010	vert	6	110	jaune
3	011	cyan	7	111	blanc

Les différents moyens de codage reposent sur un certain nombre de conventions. Par exemple, modifier l'ordre des couleurs primaires dans le codage des couleurs en vert, bleu, rouge à la place de rouge, vert, bleu modifie bien entendu les couleurs associées aux nombres. Respecter ces conventions, c'est adopter un *standard*, et cette adoption facilitera grandement les échanges de données entre les personnes ou les programmes. En particulier, si le standard est publié, on dit qu'il est *ouvert*[4], par opposition à un standard fermé, qui restreint, de manière légale ou en ne publiant pas les spécifications du standard, l'écriture d'applications compatibles qui pourraient utiliser les mêmes fichiers de données, par exemple.
Les standards de représentation et de codage des informations sont extrêmement nombreux. Pour chaque type d'objet numérique, il existe de nombreux standards : pour le texte (latin, UTF, ASCII...), pour les documents mis en page (Open Document Format, Office Open XML...), les images (JPEG, PNG...), l'audio (MP3, OGG, FLAC...), la vidéo (MPEGv2, VORBIS, H-264...).

4. Précisément, un standard ouvert est publié mais ne doit pas imposer non plus de restrictions quant à son utilisation.

2.4 Codage du texte

Les caractères sont généralement codés par des nombres, par une simple table de correspondance. Le standard le plus ancien est l'ascii, qui contient les caractères latins non accentués, les chiffres, des symboles de ponctuation, codés sur 7 bits (au total, la figure I.1 contient donc 128 symboles, caractères non imprimables compris)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
001	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
002	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Figure I.1 – Table ascii (numérotée de 0 à 7F, en héxa) tirée de Wikipédia

Le code ascii a rapidement été insuffisant, faute de caractères accentués, grecs, cyrilliques, hébreux, chinois...
Deux autres types de codage ont donc fait leur apparition :

- Les *extensions* rajoutent des caractères à la suite de la table (chaque alphabet possède alors son extension) : c'est le cas du latin-1 (iso-8859-1). Il y a des extensions pour le chinois, l'hébreu... Il faut donc indiquer quelle est l'extension utilisée pour pouvoir lire correctement le fichier.
- Unicode est une norme qui recense *tous* les caractères existant dans une table (il y en a plus de 100 000 actuellement). Les codes sont ensuite représentés dans un codage particulier comme utf-8, ou utf-16. C'est ce type de codage qu'il convient d'utiliser maintenant.

Le texte peut aussi être enrichi : changement de la taille des caractères, de la graisse, utilistion de l'italique... Là aussi, il existe une multitude de manières de représenter du texte enrichi. Nous pouvons citer le format Rtf (Rich Text Format), le Html (HyperText Markup Language), et dans une certaine mesure les formats issus de logiciels de bureautique.
Voici l'extrait d'un fichier html :

```
<html>
<body>
<h1>Algorithmique</h1>
<p>
L'<b>algorithmique</b> est l'ensemble des règles et des techniques
qui sont impliquées dans la définition et la conception
d'<a href="http://fr.wikipedia.org/wiki/Algorithme">algorithmes</a>,
c'est-à-dire de processus systématiques de résolution
d'un problème permettant de décrire les étapes vers le résultat.
En d'autres termes, un algorithme est une suite finie et non-ambiguë
d'instructions permettant de donner la réponse à un problème.
</p>
</body>
</html>
```

Le texte enrichi précédent, issu de Wikipédia, contient un paragraphe, avec un titre de niveau 1, un mot en gras, et un lien hypertexte vers une autre page Wikipédia. Tous ces éléments sont assez faciles à repérer. Le navigateur Web les interprête pour nous et affiche une page avec le texte mis en forme.

2.5 Sons

Le codage d'un son, ou d'une courbe en général, est réalisé en échantillonnant dans le temps le signal continu. Chaque échantillon est représenté par un nombre, avec une certaine précision (souvent exprimée en bits). Ce codage se nomme pcm (Pulse Code Modulation).
Selon la nature du signal ainsi numérisé, il est possible d'utiliser des codages plus économes en mémoires (compression), comme Flac (compression sans perte) ou mp3 (compression avec pertes).

Compact-disc Audio

Sur un compact-disc audio, les données ne sont pas compressés, et ce sont les échantillons qui sont directement représentés. Les enregistrements sont stéréo (deux signaux), échantillonnés à 44100 Hz (pour conserver les fréquences jusqu'à 20kHz), chaque échantillon étant représenté sur 16 bits.

2.6 Images

Un image numérique (telle que celle stockée dans les appareils photos) est une matrice de *pixels*. Chaque pixel est un point indivisible en couleur de l'image. Les éléments importants en matière de qualité de l'image sont donc : la taille de la matrice, et le nombre de bits utilisés pour coder la couleur de chacun des pixels.

Typiquement, chaque pixel est codé sur 24 bits (8 bits par composante) et un appareil photo annonçant 12 millions de pixels réalise des photographies de 4000 sur 3000 pixels.

Résolution

La *résolution* en DPI (Dots Per Inch) fait référence au nombre de pixels d'une image rapporté à la taille physique (sur un écran ou sur papier). On estime qu'une impression est très correcte (pour des besoins ordinaires) au delà de 300 DPI, c'est à dire au delà de 300 pixels par pouce. La taille maximum à laquelle pourra être imprimée une photo prise avec un appareil à 12 millions de pixels, tout en satisfaisant cette contrainte de qualité, sera donc ($1\,\text{pouce} = 2,54\,\text{cm}$) :

$$4000/300 \times 2,54 \approx 33,8\,\text{cm}$$

c'est à dire environ 25 centimètres sur 34.

Formats d'image

Si la compréhension de la numérisation d'une image est assez simple, le stockage de l'image numérique est réalisé dans un des multiples formats disponibles. Comme pour le son, ces codages ont généralement pour objectif de réduire la taille des fichiers de stockage. La compression réalisée peut être faite avec perte (c'est le cas du format JPEG) ou sans perte (c'est le cas du format PNG). Le nombre de couleurs simultanément présentes peut être de 16,7 millions (codage sur 24 bits sans palette) ou moins (comme avec le format GIF). Ainsi, le choix d'un format d'image peut dépendre du type d'image à coder (on utilise facilement JPEG pour des photos, et le format PNG est préférable pour des dessins au trait).

2.7 Codages symboliques

Avant de terminer cette section sur le codage, signalons l'existence des codages *symboliques*, que l'on peut rencontrer pour tout type d'objet (images, musique). Grossièrement, on peut dire que le codage symbolique est un codage *plus intelligent* de l'objet. Il se situe à un niveau supérieur d'abstraction et s'intéresse au *contenu* et au *sens* de l'objet plutôt qu'à son *rendu*.

Cette différence est présente aussi en dehors du monde numérique : un enregistrement musical (du son donc...) n'est pas la même chose qu'un ensemble de partitions musicales, même si l'enregistrement correspond à des musiciens jouant précisément ces partitions. La partition est alors le codage symbolique.

Cette même dualité existe dans le monde numérique. Le codage d'images que nous avons déjà évoqué est dit codage matriciel ou bitmap. Mais nous pouvons aussi décrire une image (certains types tout au moins) en détaillant les objets qui sont présents sur cette image (des cercles, des lignes etc...). Une description des objets géométriques composant un dessin est un codage *vectoriel* de l'image. Cette description a pour principales caractéristiques d'être plus économe en mémoire (dire que l'image est un disque blanc est plus rapide que détailler les 12 millions de pixels d'une photo d'un disque blanc...), et de ne pas être sensible aux problèmes de *pixelisation* (de près ou de loin, un disque blanc est toujours aussi parfait, ce qui n'est pas le cas de sa description matricielle). Les formats vectoriels usuels sont : postscript (PS), portable document format (PDF), AU, SWF, scalar vector graphics (SVG)...

De même, en ce qui concerne le son, le codage symbolique le plus utilisé est MIDI, qui détaille un morceau en pistes, chaque piste correspondant à un instrument et contenant les notes jouées, avec leur hauteur et leur durée. C'est plus ou moins l'équivalent d'une partition. On peut d'ailleurs créer de manière *automatique* une partition à partir d'un fichier MIDI, alors qu'il est peu probable qu'on puisse y parvenir à partir de l'enregistrement d'une symphonie. Le passage du symbolique vers l'échantillonné est généralement facile [5]. En revanche, l'inverse est difficile, et parfois impossible (tous les sons enregistrés ne peuvent pas être retranscrits par une partition, et lorsque c'est possible, la méthode n'est pas triviale).

5. Pour une image, cette étape s'appelle la *rasterisation* et elle est effectuée chaque fois qu'on affiche l'objet vectoriel sur un écran.

3 Langages de programmation

Science is what we understand well enough to explain to a computer.
Art is everything else we do.
Donald E. Knuth

Un langage de programmation permet d'écrire un programme. Un programme, lorsqu'il est sous la forme de code source, est un texte qui exprime un *algorithme* (nous y reviendrons), en vue de permettre l'exécution de ce dernier sur une machine.
Les langages utilisés pour programmer sont situés quelque part entre les séquences de 0 et 1 chères à la machine et le langage naturel cher à l'humain.

Pourquoi choisir un langage ?

L'informatique sur papier est possible, mais elle est moins distrayante que sur machine. De plus, la réalisation d'un programme est le moyen d'obtenir des réponses *effectives* aux problèmes qui nécessitent l'utilisation d'un ordinateur. Enfin, la programmation est une activité de création et de rigueur très formatrice, et elle aide à comprendre la manière dont fonctionnent les algorithmes.

Pseudo-code

Le pseudo-code est utilisé dans de nombreux ouvrages d'algorithmique (ci-dessous, l'exponentiation modulaire donnée dans le CLR[6], et légèrement remise en page) :

```
Exponentionation_modulaire(a,b,n)
    c←0
    d←1
    soit < b_k, b_{k-1}, ..., b_0 > la représentation binaire de b
    pour i←k à 0 faire
        c←2c
        d←(d.d) mod n
        si b_i=1 alors
            c←c+1
            d←(d.a) mod n
    retourner d
```

Il s'agit d'un algorithme rédigé dans un langage assez naturel (plutôt qu'un langage de programmation), mais qui colle tout de même aux constructions habituelles des langages (tests, boucles...). Dans ce cours, nous n'utiliserons pas particulièrement de pseudo-code. Il est cependant *tout à fait légitime* de résoudre les exercices proposés en écrivant du pseudo-code, ou même en mélangeant du code dans un langage particulier et du pseudo-code. L'inconvénient essentiel est qu'alors on ne peut pas tester nos algorithmes...

Il existe des centaines de langages différents, qui peuvent être plus ou moins spécialisés pour certaines tâches, qui permettent d'utiliser différents *paradigmes* de programmation (impératif, objet, par contraintes, logique...). Les langages peuvent être plus ou moins verbeux, plus ou moins expressifs. Certains langages sont très répandus, ou au contraire ne sont utilisés que par une poignée de personnes. Les langages peuvent être jeunes ou vieillissant, et, lorsque le côté affectif s'en mêle, ils peuvent être agréables à utiliser ou au contraire agaçants...
La tâche qui consiste à choisir un langage de programmation pour illustrer un cours d'algorithmique et d'informatique, à destination d'un public hétérogène est donc... difficile. Dans la suite, c'est le langage Python qui sera utilisé. Il est : généraliste, assez répandu (la pente étant dans le bon sens...), multi-paradigmes, assez jeune, expressif et agréable à utiliser. Il n'est cependant pas le seul à posséder toutes ces qualités.
D'un point de vue plus technique, Python est aussi un langage interprété (par opposition à un langage compilé). En partie pour cette raison, il est d'apprentissage rapide, il est portable, mais il est en revanche lent à l'exécution.
Enfin, il existe plusieurs interpréteurs Python, et l'interpréteur standard, nommé CPython, est libre, gratuit et multi plates-formes.
Voici un exemple de programme écrit en Python :

```python
def fibonacci(n):
    f = [0, 1]
    while len(f) < n + 1:
        f.append(f[-1] + f[-2])
```

6. Introduction à l'algorithmique, Cormen, Leiserson, Rivest

```
    return f[n]

def main():
    n = input("Entrez un nombre entier :")
    n = int(n)
    f = fibonacci(n)
    print("Suite de Fibonacci : U_{}={}\n".format(n, f))
```

La page 17 permet de comparer un même programme écrit en plusieurs langages.

4 Constructions typiques des langages

Cette section a pour objectifs de familiariser le lecteur avec la lecture de programmes, donnés ici en Python.
Les constructions les plus utilisées sont les variables et les calculs. Les variables servent à stocker des valeurs (ici `fare` et `celc`), et les calculs sont réalisés très simplement en utilisant par exemple les opérateurs arithmétiques habituels.

```
fare = float(input("Température en degrés Fahrenheit "))
celc = (fare - 32) / 1.8
print("En degrés Celcius, cela fait : ", celc)
```

Égalité $\neq$ affectation

Attention à ne pas confondre le symbole `=`, utilisé comme symbole *d'affectation* dans la plupart des langages avec le $=$ mathématique qui indique que deux valeurs sont égales. L'équation mathématique $a = a + 1$ est par exemple (dans la plupart des contextes) sans intérêt, alors que la ligne de programme `a=a+1` indique qu'il faut rajouter 1 au contenu de `a`. Non seulement ce n'est pas absurde, mais c'est une écriture très courante.

La plupart des programmes n'ont pas un unique fil d'exécution possible. Selon les valeurs données en entrées, ou selon le résultat de certains calculs, une chose plutôt qu'une autre doit être faite (par exemple, pour un programme de recherche de racines d'un polynôme de degré 2, après que l'utilisateur a entré les coefficients, la portion de programme à exécuter ne sera pas la même selon que le discriminant est positif, négatif ou nul).
Dans l'exemple suivant, selon que la température est en dessous de $0°C$ ou non, un message différent sera affiché :

```
fare = float(input("Température en degrés Fahrenheit "))
celc = (fare - 32) / 1.8
print("En degrés Celcius, cela fait : ", celc)
if celc > 0:
    print("Ça va, il ne va pas geler...")
else:
    print("Tous aux abris, il va faire froid...")
```

Lors de la conception d'un programme un peu long, il est indispensable de le structurer en parties indépendantes. Il est très utile de commencer à prendre cette habitude même avec des programmes courts. Dans l'exemple qui suit, nous avons écrit une fonction, nommée `conversion`, qui convertit une température, donnée en degrés Fahrenheit en une température exprimée en degrés Celcius. La fonction est ensuite utilisée dans le programme principal :

```
# Fonction
def conversion(f):
    c=(f - 32) / 1.8
    return c

# Programme principal
fare = float(input("Température en degrés Fahrenheit "))
celc = conversion(fare)
print("En degrés Celcius, cela fait : ", celc)
```

Le propre de l'ordinateur est de ne pas se plaindre si on lui demande de recommencer de nombreuses fois les mêmes opérations (avec quelques petites variantes). La structure de boucle permet d'exprimer les répétitions et nous l'utilisons ci-dessous pour afficher une table de conversion de degrés Fahrenheit vers degrés Celcius (en vue de l'imprimer par exemple).

```
# Fonction
def conversion(f):
    c = (f - 32) / 1.8
    return c

# Programme principal
fare = 0
while fare < 100:
    celc = conversion(fare)
    print(fare, " / ", celc)
    fare = fare + 10
```

Toujours les mêmes ingrédients

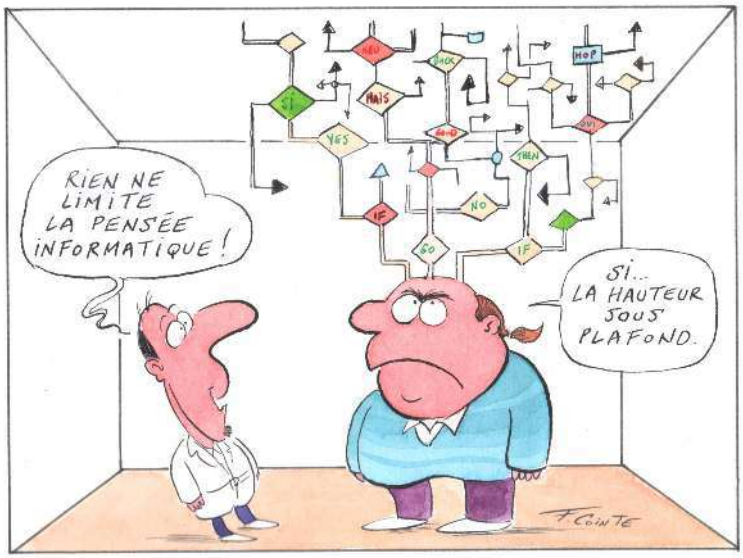
En programmation impérative procédurale (la majorité des types de programmation que vous rencontrerez), les même ingrédients se retrouvent toujours :

- séquences d'opérations (on les donne dans l'ordre)
- branchement conditionnel (c'est le test)
- itération (c'est la boucle)
- exécution d'une procédure ou fonction

Ainsi, que vous soyez amenés à programmer en Python, Matlab, C, Fortran ou autre, l'apprentissage d'un quelconque de ces langages accélère l'apprentissage des autres, de la même manière que connaître plusieurs langues étrangères facilite l'apprentissage de nouvelles.

Chapitre II

Algorithmique et programmation



Avec l'aimable autorisation de François Cointe : http://www.fcointe.com

5 À quoi sert un algorithme ?

L'algorithmique est bien plus ancienne que l'informatique, que l'ordinateur, et que le langage Python, utilisé dans ce support de cours.
Les exemples les plus anciens et célèbres sont :

- les calculs d'impôts Babyloniens (il y a 4000 ans)
- le calcul du plus grand diviseur commun (Euclide, vers −350)
- les première méthodes de résolution systématique d'équations (Al Khawarizmi, IX^{e} siècle)

Les algorithmes étaient donc d'abord utilisés «à la main».
Aujourd'hui on entend généralement par algorithmique la réflexion préliminaire à l'écriture d'un programme d'ordinateur, c'est à dire la recherche d'une **méthode systématique** et **non ambiguë** pour **résoudre un problème**. C'est la **partie conceptuelle** de la programmation, l'**abstraction**[1] d'un programme d'ordinateur.
L'algorithmique est parfois considérée comme une branche des mathématiques, parfois comme une branche de l'informatique. Les notions d'algorithme, puis d'ordinateur ont été formalisées dans les années 30 et 40 par : Kurt Gödel, Alan Turing, John von Neumann...
Avoir des notions en algorithmique permet de développer soi-même des programmes, et/ou d'avoir une discussion constructive avec une équipe de développeurs. Les compétences en algorithmique font aussi partie du bagage minimum d'un scientifique, qu'il soit technicien, chercheur, ingénieur ou simplement curieux.

1. La même manière de trier une liste, d'ajouter deux nombres ou de compresser un fichier peut s'exprimer dans différents langages de programmation. Il existe donc un objet abstrait, qui s'incarne dans ces programmes écrits dans différents langages. C'est cet objet que l'on appelle un algorithme – extrait de *Introduction à la Science Informatique*

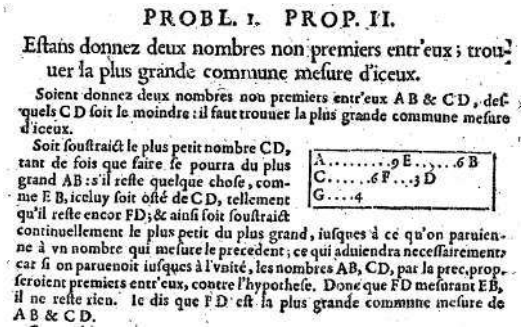
Obtenir plus d'informations

Vous trouverez plus d'information générales sur l'algorithmique sur la page *Culture Info* :

https://deptinfo-ensip.univ-poitiers.fr/ENS/doku/doku.php?id=publish:cultureinfo

En particulier, vous êtes invités à écouter le *Podcast interstices : les algorithmes.*

6 Algorithme d'Euclide



PROBL. I. PROP. II.

Eſtans donnez deux nombres non premiers entr'eux ; trouuer la plus grande commune meſure d'iceux.

Soient donnez deux nombres non premiers entr'eux A B & C D, deſquels C D ſoit le moindre : il faut trouuer la plus grande commune meſure d'iceux.

Soit ſouſtraict le plus petit nombre C D, tant de fois que faire ſe pourra du plus grand A B : s'il reſte quelque choſe, comme E B, iceluy ſoit oſté de C D, tellement qu'il reſte encor F D ; & ainſi ſoit ſouſtraict continuellement le plus petit du plus grand, iuſques à ce qu'on paruienne à vn nombre qui meſure le precedent ; ce qui aduiendra neceſſairement : car ſi on paruenoit iuſques à l'vnité, les nombres A B, C D, par la prec. prop. ſeroient premiers entr'eux, contre l'hypotheſe. Donc que F D meſurant E B, il ne reſte rien. Ie dis que F D eſt la plus grande commune meſure de A B & C D.

A.....9 E.....6 B
C.....6 F.....3 D
G....4

FIGURE II.1 – Les éléments, livre VII, édition de 1632

Commençons par un exemple historique célèbre : l'algorithme d'Euclide.

Algorithme d'Euclide

Étant donnés deux entiers, retrancher le plus petit au plus grand et recommencer jusqu'à ce que les deux nombres soient égaux. La valeur obtenue est le plus grand diviseur commun.

Exercice

Appliquez l'algorithme d'Euclide (à la main) aux nombres 133 et 49

Voici les différentes étapes vers la rédaction propre d'un algorithme. Nous partons de l'idée de départ et la raffinons par étapes...

Étape 1 : Écrire proprement l'idée

Prendre les deux nombres et, tant qu'ils ne sont pas égaux, retirer le plus petit au plus grand.

Étape 2 : Décrire précisément les étapes

```
a,b : deux nombres
Répéter tant que a et b sont différents :
    si le plus grand est a :
        les deux nombres deviennent a−b et b
    sinon (b est donc le plus grand) :
        les deux nombres deviennent a et b−a
le pgcd est a
```

Étape 3 : Écrire un algorithme formel

```
fonction pgdc(a,b : entiers)
    repeter tant que a≠b
        si a>b
            a=a−b
        sinon
            b=b−a

    retourner a
```

Étape 4 : Écrire un programme

```
def pgdc(a, b):
    while a != b:
        if a > b: a = a - b
        else:     b = b - a
    return a
```

Finalement, c'est quoi la programmation ?

Programmer est l'activité qui consiste à :

- **traduire** des algorithmes dans un langage de programmation :
  - afin d'obtenir des réponses effectives (souvent numériques) à des problèmes ;
  - afin de se distraire...
- corriger des erreurs dans un programme ;
- rester calme...

Voici l'algorithme du pgdc présenté dans plusieurs langages[2].

```
# CODE PYTHON
def pgdc(a,b):
    while a!=b :
        if a>b : a=a-b
        else   : b=b-a
    return a
```

```
% CODE OCTAVE
function r=pgdc(a,b)
  while (a~=b)
    if (a>b) a=a-b;
    else b=b-a;
  end
  r=a;
```

```
// Code Java
public static int pgdc(int a, int b) {
    while (a!=b) {
      if (a>b) a=a-b;
      else b=b-a;
    }
    return a;
}
```

```
(* Code OCaml *)
let rec pgcd(a, b) =
  if a = b then a else
    if a > b then pgcd(a-b, b)
    else pgcd(a,b-a)
  ;;
```

```
/* CODE C */
int pgdc(int a,int b)
{
  while (a != b) {
    if (a>b) a=a-b;
    else b=b-a; }
  return a;
}
```

```
# CODE RUBY
def pgdc(a,b)
    while a!=b
        if a>b then a=a-b
        else b=b-a end
    end
    a
end
```

```
; Code Scheme
(define (pgdc a b)
  (cond
    ((< a b) (pgdc a (- b a) ))
    ((> a b) (pgdc (- a b) b ))
    (else a)
  )
)
```

```
Rem Code OOBasic
Function Pgdc(ByVal a As Integer ,
        ByVal b As Integer) As Integer
 Do While a<>b
   If a>b Then
      a=a-b
   Else
      b=b-a
   EndIf
 Loop
 Pgdc=a
End Function
```

7 Poser les problèmes

Résoudre un problème nécessite :
- que l'énoncé soit précis ;
- qu'on le comprenne parfaitement ;
- ...

Autant que possible, il faut prendre l'habitude de préciser, de préférence par écrit, quelles sont les entrées et les sorties du problème (ou des sous-problèmes, comme nous le verrons plus tard).
Voici quelques exemples :

Problème : Puissance (élever un réel à une certaine puissance)

Entrées un réel x, un entier n

Sortie le réel x^n

2. Notons qu'une implémentation plus efficace n'enlève pas le plus petit nombre au plus grand, mais calcule le reste de la division entière du plus grand nombre par le plus petit. Les fonctions Scheme et OCaml sont un peu différentes des autres et utilisent la récursivité plutôt que les boucles.

Problème : Facteurs premiers

Entrées un entier $n > 1$

Sortie liste des facteurs premiers de n

Problème : Calculer le plus grand diviseur commun

Entrées deux entiers `a` et `b`

Sortie un entier, pgdc de `a` et `b`

8 Types, affectations, expressions

8.1 Types simples

Le type d'une donnée caractérise sa nature (est-ce que c'est un entier, un nombre à virgule une couleur, une liste... ?) et par voie de conséquence les opérations qu'il est possible de lui appliquer (on peut ajouter des nombres, mais peut être pas des couleurs).
Les types dans un langage informatique dépendent fortement du codage utilisé pour représenter les données. En algorithmique, on peut parfois se permettre d'utiliser des types «mathématiques» qui n'ont pas d'implémentation réelle en machine.

Cas des entiers et des réels

Les entiers, par exemple, ont une taille limitée en C (quelques octets). Cette taille est limitée à la mémoire disponible en Python (l'entier maximum est considérablement grand). L'ensemble des entiers «mathématiques» n'est en revanche pas borné.

Les réels, que l'on manipule en mathématique n'existent ni en Python, ni en C, où il sont remplacés par les nombres à virgule flottante (ou flottants). En conséquence, tous les calculs sur ordinateur utilisant les flottants sont *intrinsèquement faux*.

Voici quelques exemples de types simples :

Nom commun	Python	C	
Entier	`int`	`int`	en C, un entier a une taille limite assez faible
Réels	`float`	`float`,`double`	Les calculs en flottants sont approchés
Booléen	`bool`	`int`	
Caractère		`char`	En Python, il n'y a pas de type caractère

8.2 Variables

Les variables peuvent être vus comme des espaces de stockage (métaphore du tiroir) ou comme des noms associés à des objets (étiquettes). Le modèle qui correspond à Python est celui de l'étiquette. Une variable possède plusieurs caractéristiques :

identificateur (nom) Il doit être bien choisi, surtout dans un programme long. Un identificateur ne comporte aucun espace ou caractère spécial et ne peut pas être un mot clé. Chaque langage dispose de règles (différentes) très précises décrivant la syntaxe des identificateurs

type Dans certains langages, le type d'une variable peut changer en cours d'exécution du programme (Python, Ruby). Dans d'autres, le type est fixé une fois pour toutes (Java, C).

valeur C'est l'objet/valeur désigné par la variable.

portée On ne peut utiliser le nom d'une variable que dans la fonction ou la procédure (en réalité le bloc) qui la contient [3]. La durée de vie de la variable correspond à la durée d'exécution de la fonction ou de la procédure (du bloc). Sans ce principe, les programmes longs deviendraient incompréhensibles et difficiles à déverminer.

Attention à l'affectation

Nous l'avons déjà mentionné, mais nous rappelons qu'il ne faut pas confondre l'affectation et l'égalité. L'affectation est une opération *typiquement* informatique. En Python elle permet d'affecter un nouveau nom (celui indiqué à gauche du `=`) à un objet (indiqué à droite du `=`). L'opérateur d'affectation **n'est donc pas symétrique**.

Voici un exemple de session shell qui calcule un prix TTC :

3. En Python, voir la règle LEGB mentionnée plus loin.

```
>>> a = 19.6
>>> sommeht = 245
>>> sommettc = sommeht * (1 + a / 100)
>>> print(sommettc)
```

Exercice

La session shell suivante affiche deux prix HT et TTC : 1000 et 1196, puis 2000 et 1196... Il y a manifestement une erreur. La comprenez-vous ?

```
# Attention, ce programme comporte
# une erreur de conception
>>> sommeht = 1000
>>> sommettc = sommeht * 1.196
>>> print(sommeht, sommettc)
>>> sommeht = sommeht * 2
>>> print(sommeht, sommettc)
```

L'affectation en Python

En Python, l'affectation a un sens un peu particulier qui pourra vous conduire à de mauvaises surprises. Nous y reviendrons par la suite, mais les impatients peuvent partir d'un principe que les données ont une existence en mémoire et que les variables sont des *références* vers ces données en mémoire.

L'opération d'affectation : `x=DATA` finalise la création en mémoire de l'objet `DATA`, puis stocke dans `x` une référence vers cet objet.

En particulier, une affectation en Python ne **duplique** jamais un objet, mais **donne** simplement un nouveau nom à un objet existant.

Le comportement observé est la plupart du temps le même que celui des autres langages... mais pas toujours.

8.3 Expressions

Nous utiliserons une définition «intuitive» des expressions, en retenant que c'est une portion de code, plus ou moins longue, qui peut être évaluée. La valeur obtenue peut par exemple être affectée à une variable.

- `3*i+5` : ok si `i` est un nombre
- `pgdc(105,i)+3` : ok si `i` est un entier, et si `pgdc` est une fonction retournant un nombre.

On utilise les notations les plus classiques possibles : `+`, `-`, `*`, `/`, `%` (modulo), `**` (puissance)

Division entières

Attention, en Python 3 (mais pas en Python 2), la division entre deux entiers, notée `/`, donne un flottant (même si la division tombe juste). La division entière est en revanche notée `//`. Ce comportement est différent de ce qu'on trouve dans la plupart des langages, pour lesquels c'est le type des opérandes qui fixe le type du résultat.

8.4 Listes

Un tableau, tel que ceux utilisés en C par exemple, permet de stocker selon une ou plusieurs dimensions des données homogènes (on parle de tableau d'entiers par exemple).
Si le type `array` existe aussi en Python (module `array`), on utilise plus volontiers dans les langages de scripts modernes le type `list` qui permet de stocker des objets hétérogènes (voire d'autres listes).
Voici un exemple d'utilisation de listes :

```
    # Créer des listes
>>> l1 = []
>>> l2 = [1, 3, 2, 4]
    # Mettre à jour un élément
>>> l2[3] = 45
    # Accéder à un élément
>>> print(2 * l2[1])
    # Attention !!!
>>> l1[0] = 5
    # Rajout d'élément
```

```
>>> l1.append(56)
>>> l2.append('Fin')
    #Connaître le nombre d'éléments
>>> len(l1)
>>> len(l2)
```

8.5 Tuples

Les *tuples*[4] de Python sont des listes *non modifiables*[5].
Il n'est pas strictement nécessaire d'utiliser des tuples. Mais certaines méthodes ou fonctions Python en renvoient et il faut donc connaître leur existence.

9 Fonctions et procédures

9.1 Fonctions

Les *fonctions* et *procédures* sont la pierre angulaire de la *programmation procédurale*.
Il y aura deux points fondamentaux à retenir dès la fin de lecture de cette sous-section :

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

Commençons par un exemple en Python :

```
def calculttc(val, taux):
    """
    Calcule le montant ttc, connaissant la somme
    hors taxe (val) et le taux (par ex 19.6)
    """
    ttc=val * (1 + taux / 100)
    return ttc
```

Instruction `return`

Que ce soit en langage algorithmique, en C, en Octave ou en Python, l'instruction retourner (**`return`**) *fait se terminer la fonction ou la procédure* (dans le cas d'une procédure, **`return`** n'est pas suivi d'une valeur), même si cette instruction est exécutée avant la fin du texte de la fonction.

Une fois la fonction `calculttc` lue par le shell, il est possible de l'utiliser ainsi :

```
>>> calculttc(1000, 19.6)
=> 1196
```

Notons qu'il convient de *bien séparer* la définition et l'utilisation de la fonction :

définition : on indique comment «marche» la fonction et comment on s'en sert (déclaration). La fonction n'est pas utilisée (exécutée), mais juste «lue» pour être connue de l'interpréteur.

utilisation : on exécute de manière effective le code de la fonction *pour des valeurs d'entrées fixées* (1 000 et 19, 6 dans l'exemple).

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

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

4. Nous devrions les appeler n-uplets en français, mais c'est le mot anglais qui est systématiquement repris dans les ouvrages sur Python.
5. Cela signifie que chaque élément du tuple devra toujours référencer le même objet (mais cet objet, s'il est lui-même modifiable pourrait changer)...

9.2 Fonctions dans d'autres langages

La façon d'écrire les fonctions, méthodes et procédures varie selon les langages. Une manière plus algorithmique et francisée d'écrire pourrait être :

fonction calculttc (val ,taux : réels) : réel
    ttc : réel
    ttc←val∗(1+taux/100)
    retourner ttc

Voici d'autres exemples, pour comparaison :

```
// En langage C
float calculttc(float val,float taux)
{
  float ttc;
  ttc=val*(1+taux/100);
  return ttc;
}
```

```
# En Ruby
def calculttc(val,taux)
  val*(1+taux/100)
end
```

```
// En Octave
function ttc=calculttc(val,taux)
  ttc=val*(1+taux/100)
```

9.3 Procédures

Il ne faut pas confondre procédure et fonction, même si leur définition, en Python par exemple, est similaire :

- Une fonction calcule. Elle *vaut* quelque chose (exemples : `pgdc` et `calculttc`)
- Une procédure n'a pas de valeur[6], mais elle fait/modifie quelque chose (exemple : afficher à l'écran, modifier un objet). On dit qu'elle a un effet de bord (elle modifie quelque chose qui est en dehors de la procédure).

Même si la différence fonction/procédure n'est pas syntaxique (au niveau de la ligne de prototype) en Python, elle est très importante :

```
# Ceci est une fonction
def foncttc(val, taux) :
    ttc = val * (1 + taux / 100)
    return ttc
```

```
# Ceci est une procédure
def procttc(val, taux) :
    ttc = val * (1 + taux / 100)
    print(ttc)
```

On pourrait penser que la fonction `foncttc` et la procédure `procttc` sont équivalentes[7]. Ce serait une erreur. La fonction est plus générale. On ne peut pas écrire par exemple :

```
    # La ligne suivante ne fonctionne pas
>>> print("Double du prix ttc : ", procttc(100, 19.6) * 2)
```

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

Effets de bords

Autant que possible, **on souhaite qu'une fonction n'ait pas d'effet de bord**, d'aucune sorte :

- pas de modification d'un objet qui n'appartient pas à la fonction (variable globale par exemple, ou objet passé par référence en paramètre) ;

6. En Python, une procédure renvoie la valeur spéciale `None`.
7. Si on entre `procttc(100,7)` ou `foncttc(100,7)` dans le shell, on verra une réponse correcte s'afficher (quoique un peu différemment) dans les deux cas...

— pas d'entrées/sorties ;
— ...
⇒ on évite donc les affichages dans une fonction (ce n'est pas interdit... c'est *déconseillé*) : une fonction calcule un résultat, le renvoie, et on l'affiche ensuite, hors de la fonction.

9.4 Prototype d'une fonction

Le prototype est la première ligne de la fonction :

fonction calculttc (val , taux : réels) : réel

Il peut indiquer 4 choses :

1. le nom de la fonction ou de la procédure : `foncttc`
2. les paramètres (les entrées du problème) : `(val,taux : réels)`
3. les éventuels types de sortie (dans le cas d'une fonction) : `réel`
4. si l'on a affaire à une fonction ou à une procédure (on le voit si le type des données de sortie est mentionné par exemple, ou parce que le mot clé est différent (`fonc` ou `fonction` pour une fonction et `proc` ou `procédure` sinon).

Ce prototype est très abrégé en Python :

```
def foncttc(val, taux)
```

Le prototype Python n'indique pas les types des entrées, et nous ne pouvons pas savoir si nous avons affaire à une fonction ou à une procédure.

9.5 Variables locales

Les variables locales sont les variables qui n'existent que dans le corps de la fonction ou procédure (dans une certaine mesure, les paramètres peuvent être assimilés aux variables locales). Voici un exemple de fonction avec un paramètre `u` et une variable locale `v` :

```
def aucarre(u):
    v = u ** 2
    return v
```

Pour utiliser la fonction qui précède, il est inutile de connaître les noms de variable `u` et `v`. Ce qui est *important*, c'est que la fonction *prend en paramètre un nombre et renvoie un nombre* (le carré du premier) :

```
>>> a = 9
>>> b = aucarre(a)
>>> print(a, b)
9 81
```

La notion de *portée des variables* (variables locales, globales) est souvent mal comprise au début : elle peut être perçue comme une difficulté, voire un défaut, alors qu'elle simplifie grandement le travail du programmeur.

Local, englobant, global et built-in

Les auteurs anglophones conseillent de retenir la règle LEGB, acronyme qui permet de retenir l'ordre de résolution des noms de variable dans Python : local, englobant, global et *built-in*. Les variables *locales* à une fonction sont celles définies *dans* la fonction ainsi que ses paramètres. Les variables du bloc englobant d'une fonction sont celles dont la portée contient la fonction. Les variables globales sont celles déclarées au niveau du fichier python. Enfin, *built-in* regroupe ce qui est défini à l'exécution de l'interpréteur.

Dans un premier temps, il est nécessaire de :
— ne pas utiliser de variable globale (ce qui est une bonne chose de toutes façons) ;
— bien comprendre qu'une variable *locale* à une fonction n'existe pas en dehors de cette fonction.

9.6 Conseils avisés

Malheureusement...

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as

cleverly as possible, you are, by definition, not smart enough to debug it.

Brian Kernighan

Tiré de **Math-Info** de *Raymond Séroul*, InterÉditions

On ne mâche pas un chewing-gum en montant un escalier :
Ce proverbe – tiré d'une campagne électorale américaine – exprime une idée de bon sens : on ne fait bien qu'**une seule chose à la fois**. C'est pour cela qu'on **découpe un programme en procédures** et en modules indépendants.

Nommez ce que vous ne connaissez pas encore :
Ce proverbe s'applique chaque fois que vous rencontrez un sous-problème à l'intérieur d'un problème. Refusez de résoudre le sous-problème, laissez-le de côté et avancez. Comment ? En **donnant un nom à ce que vous ne connaissez pas encore** (du code, une fonction). Cela vous permettra de terminer le travail (i.e. le problème) en cours. Pour le sous-problème, voyez le proverbe suivant.

Demain, ce sera mieux ; après-demain, ce sera encore mieux :
[...] vous constatez un blocage provoqué par **l'apparition d'un nouveau problème** à l'intérieur du problème que vous essayez de résoudre : **appliquez le proverbe précédent** [...] apprenez à distinguer l'essentiel de l'accessoire ; **ne vous noyez pas prématurément dans les détails** [...] Vous l'avez certainement pratiquée en mathématique : «Je vais d'abord prouver mon théorème en admettant provisoirement les lemmes 1, 2 et 3». [...]

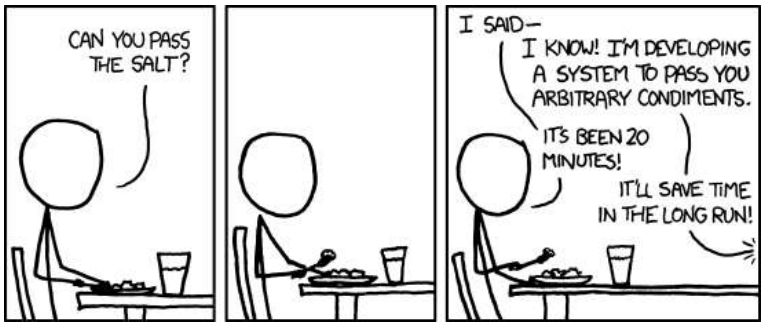
Tester un programme

Une fois le programme écrit, il faut vérifier qu'il est juste. Pour cela, on le teste sur des valeurs particulières. Le choix de ces valeurs est important. Il faut :

- tester avec des entrées pour lesquelles on connaît la sortie correcte ;
- tester de manière extensive, afin que chaque portion de code soit exécutée ;
- penser aux cas extrêmes qui posent parfois des problèmes.

À retenir sur les fonctions

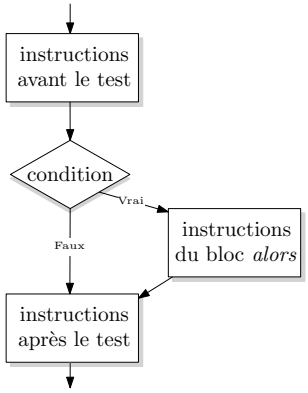
1. Une fonction *vaut* quelque chose, une procédure *fait* quelque chose.
2. Le *prototype* (déclaration) d'une fonction suffit (normalement) à savoir *ce qu'elle calcule* et *comment on l'utilise*, mais pas comment elle le calcule.
3. On écrit des fonctions et des procédures pour *ne plus avoir à se soucier de la façon de régler le problème qu'elles traitent*. En conséquence, être obligé de retoucher le code d'une fonction pour un problème particulier indique généralement que la fonction a été mal conçue au départ.
4. Généralement, lorsqu'on a terminé l'écriture d'une fonction et qu'on l'a testée, *on souhaite ne plus avoir à revenir dessus* et ne plus devoir la modifier.



http://xkcd.org

10 Conditions

Une mère dit à son fils : «Va au marché et achète une bouteille de lait. S'il y a des oeufs, prends-en six.»
Le fils revient avec six bouteilles de lait.
Sa mère lui demande : «Mais pourquoi as-tu ramené six bouteilles de lait ?»
Le fils répond : «Parce qu'il y avait des oeufs !»
Le cyberblog du coyote



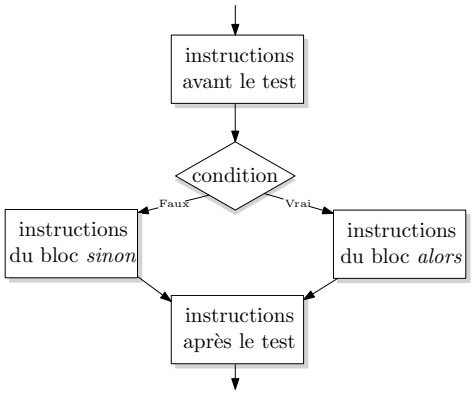

```
<instructions avant le test>
if <condition>:
    <instructions du bloc "alors">
<instructions après le test>
```

Nous souhaitons demander un nombre à l'utilisateur et afficher un message particulier si ce nombre est un multiple de 13 :

```
n = int(input('Entrez un nombre :'))
if n % 13 == 0:
    print(n, 'est divisible par 13')
    print('Bravo', n)
print("Merci d'avoir participé")
```

Exercice

Identifiez chaque partie du programme qui précède avec les blocs correspondant de l'organigramme.



On cherche à traduire un choix, une alternative.

```
<instructions avant le test>
if <condition>:
    <instructions du bloc "alors">
else:
    <instructions du bloc "sinon">
<instructions après le test>
```

La suite de Collatz est définie ainsi :

$$u_{n+1} = \begin{cases} u_n/2 \text{ si } u_n \text{ est pair} \\ 3u_n + 1 \text{ sinon} \end{cases}$$

Pour chaque valeur de u_0 choisie, on obtient une nouvelle séquence de nombres. Écrivons une fonction qui calcule u_{n+1} en fonction de u_n.

```
fonction collatz (u : entier) : entier
    res : entier
    si u est pair
        res ←u/2
    sinon
        res ←3*u+1
    retourner res
```

```
def collatz(u):
    if u % 2 == 0:
        res = u // 2
    else :
        res = 3 * u + 1
    return res
```

Exercice

Identifiez chaque partie du programme qui précède avec les blocs correspondant de l'organigramme.

Opérateurs booléens

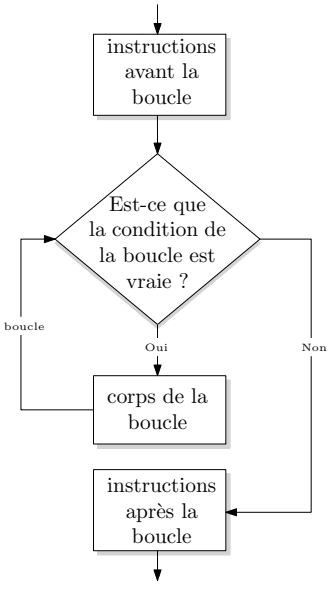
Ce sont les classiques : **and**, **or**, **not**.
Attention aux priorités :

`True or True and False` vaut `True`
`(True or True)and False` vaut `False`
`True or (True and False)` vaut `True`

11 Boucles

Une mère dit à son fils :
«Va au supermarché acheter une bouteille de lait. Et tant que tu y es, prends des oeufs.»
Il ne revint jamais.
linuxfr.org

11.1 Boucles Tant Que




```
<instructions avant la boucle>
while <condition>:
  <corps de la boucle>
<instructions après la boucle>
```

Nous souhaitons faire un programme de jeu qui propose à l'utilisateur de deviner un nombre entre 1 et 100 en faisant des propositions. À chaque proposition, le programme lui indique si le nombre proposé est trop petit ou trop grand. Le jeu s'arrête lorsque le nombre est découvert.

```
import random
nb = random.randint(1, 100)
ch = -1
while nb != ch :
  ch = int(input('Entrez un nombre '))
  if ch < nb : print('Trop petit')
  if ch > nb : print('Trop grand')
print('Bravo')
```

Exercice

Identifiez chaque partie du programme qui précède avec les blocs correspondant de l'organigramme.

Voici un autre exemple de boucle : la suite de collatz, que nous avons récemment rencontré a la propriété, quel que soit son nombre de départ, de toujours terminer sur le cycle 1, 2, 4. Ce fait a été vérifié expérimentalement jusqu'à de

très grandes valeurs, mais a pour l'instant le statut de conjecture car la preuve semble échapper aux mathématiques actuelles.
Voici une fonction qui renvoie une liste contenant la suite issue d'un nombre n, jusqu'à ce que la valeur 1 soit atteinte. Cette liste s'appelle le vol n :

```
def suite(depart):
    s = []
    val = depart
    while val != 1 :
      s.append(val)
      val = collatz(val)
    s.append(1)
```

Exercice

Identifiez chaque partie du programme qui précède avec les blocs correspondant de l'organigramme. Profitez de la présence d'une machine près de vous pour vous émerveiller devant la longueur du vol 270. Quel est son point culminant ?

11.2 Boucles Répéter Pour et itérateurs

Beaucoup de langage offrent un autre type de boucle, permettant de gérer un compteur. Voici un exemple qui affiche des tables de multiplication :

Répéter pour i de 1 à 10 :
  Répéter pour j de 1 à 10 :
    écrire (i ,'x', j ,'=', i*j)

Python n'est pas en reste pour ce type de boucle, même si la construction offerte est plus générale que la simple boucle avec compteur.
La traduction de l'algorithme qui précède en Python donnerait :

```
for i in range(1, 11):
  for j in range(1, 11):
    print(i, 'x', j, '=', i * j)
```

La boucle **for** de Python permet en fait d'itérer sur n'importe quel objet *itérable*. Or l'objet `range(a,b)` est itérable et «contient» les entiers de `a` à `b-1` inclus. C'est pourquoi dans l'exemple qui précède `i` prend successivement toutes les valeurs de 1 à 10.
Avec la boucle **for** de Python, il est possible de traverser toutes sortes d'objets :

```
liste=(5, 10, 15, 23, 13, 29)
for nombre in liste:
   print('Le nombre est',nombre)
for lettre in 'Hello World':
   print(lettre)
for i,l in enumerate('Hello World') :
   print(i, l)
for mot in 'Hello World'.split():
   print(mot)
```

1. Dans la première boucle, `nombre` prendra successivement toutes les valeurs de la liste.
2. Une chaîne est itérable. Dans la seconde boucle, `lettre` vaudra successivement chaque lettre de la chaîne de caractères
3. La fonction **enumerate** sur un itérable renvoie un nouvel itérable dont les éléments sont des tuples à 2 éléments : le premier est un numéro d'ordre et le second l'élémént dans l'itérable de départ. Par exemple `enumerate('Hello')` *correspond* à `[(0, 'H'), (1, 'e'), (2, 'l'), (3, 'l'), (4, 'o')]`. Cette boucle aurait pu être écrite :

   ```
   for t in enumerate('Hello World'):
       print(t[0], t[1])
   ```

La première écriture dépaquette le tuple `t` en `i` et `l` à la volée.

4. la méthode `split` renvoie une liste contenant chaque mot de la chaîne. Dans la quatrième boucle, `mot` vaudra donc successivement `'Hello'` puis `'World'`

Inspecter les objets itérables En première lecture, vous pouvez omettre ce paragraphe.
Depuis Python 3, il n'est pas toujours immédiat d'inspecter les valeurs d'un objet itérable :

```
>>> s = "analphabète"
>>> l = enumerate(s)
>>> l
<enumerate object at 0x7feecc10cb40>
```

Pour connaître les valeurs de `l`, on peut le transformer en liste :

```
>>> list(l)
[(0, 'a'), (1, 'n'), (2, 'a'), (3, 'l'), (4, 'p'), (5, 'h'), (6, 'a'), (7, 'b'), (8, 'è'),
  (9, 't'), (10, 'e')]
```

Mais attention, l'objet `l` est *générateur* qui ne peut être traversé qu'une fois (et le transformer en liste implique qu'on le traverse) :

```
>>> list(l)
[]
```

Pour à nouveau accéder aux valeurs, il suffit de créer un nouveau générateur :

```
>>> l = enumerate(s)
>>> list(l)
[(0, 'a'), (1, 'n'), (2, 'a'), (3, 'l'), (4, 'p'), (5, 'h'), (6, 'a'), (7, 'b'), (8, 'è'),
  (9, 't'), (10, 'e')]
```

12 Commentaires

Les commentaires sont signalées par un `#` en Python. Tous les caractères qui suivent ce symbole sont ignorés par l'interpréteur.
Les commentaires servent à aider le lecteur humain à comprendre le programme. Il n'en faut pas trop, et ils doivent être judicieux (on ne commente pas ce qui est évident).

Bon

```
# Met dans s les diviseurs
# stricts de n
s={1}
for i in range(2,n) :
    if n % i == 0 :
        s.add(i)
```

(très) Mauvais

```
# Initialise s à 1
s={1}
# Pour i variant de 2 à n-1
for i in range(2,n) :
    # si le reste de la
    #division de n par i est nul
    if n % i == 0 :
        # ajouter i à s
        s.add(i)
```

13 Notation pointée

Bien que ce support de cours n'ait pas pour but d'enseigner la programmation orientée objets, il est nécessaire d'en comprendre certaines notations afin de bien utiliser Python.
Un objet est une entité composée d'attributs (typiquement des valeurs) et de méthodes (les actions que l'objet peut effectuer). C'est le cas de la tortue du module `turtle`. Un objet de type `Turtle` *encapsule* une position, une orientation, la taille du trait à tracer, la couleur à utiliser etc... Tout ceci est stocké dans l'objet. On peut par exemple disposer de deux tortues, l'une écrivant en rouge et l'autre en bleu.
Les méthodes sont les actions que peut effectuer la tortue : avancer (`forward`), tourner à droite (`right`)...

Les objets offrent un plus grand niveau d'abstraction : Si la tortue est en (0,0), orienté selon l'angle 45, et que nous lui ordonnons d'avancer de 10, le calcul de la nouvelle position est fait *dans* l'objet : Nous donnons des ordres de haut niveau et l'objet gère ses détails interne.
Les ordres donnés à des objets (les appels de méthodes donc) utilisent la notation pointée. Pour faire avancer une tortue `t1` de 50 par exemple, nous indiquerons `t1.forward(50)` qui signifie : appeler la méthode `forward` de l'objet `t1` et lui transmettre le paramètre `50`. La fonction **`help`** de python, lorsqu'on lui donne un objet en paramètre, indique ainsi l'ensemble des attributs et des méthodes de l'objet en question.
Voici un exemple d'utilisation du module `turtle` :

```
>>> from turtle import *
>>> t1 = Turtle()
>>> t2 = Turtle()
>>> t1.forward(50)
>>> t1.color('blue')
>>> t2.color('red')
>>> t2.right(90)
>>> t1.left(90)
>>> t2.forward(100)
>>> t1.forward(100)
```

Rappelons que nous avons déjà utilisé la notation pointée :

```
l = [1, 2, 3, 4]
l.append(5) # ajoute 5 à la liste l
...
```

Dans l'exemple qui précède, `l` est une référence vers un objet, instance de la classe **`list`**. Cet objet contient des données (les références vers les objets stockés dans la liste, et d'autres choses encore) ainsi que des méthodes, que l'objet peut appliquer.
La programmation orientée objets ne se résume pas à ça bien sûr, mais ce qui précède est l'essentiel à savoir pour *utiliser* les objets.

14 Affectation, mode de passage des paramètres

Par valeur ou par adresse

Dans la plupart des langages, on différencie le passage des paramètres par valeur et par adresse. Dans le premier cas, c'est une copie du paramètre qui est communiquée à la fonction. Dans le second, c'est l'objet original, si bien qu'une modification de l'objet passé en paramètre dans la fonction modifie effectivement l'objet original.

Lorsqu'on écrit un algorithme, c'est bien souvent le contexte et le type de valeurs manipulées (on fait donc appel à l'expérience du lecteur) qui permet de connaître le type de passage de paramètres envisagé.

Dans ce qui suit, nous allons nous appuyer spécifiquement sur le mécanisme de Python.

Intéressons-nous au problème suivant : Si on envoie une variable en paramètre à une fonction ou une procédure, et qu'on la modifie dans la fonction, est-elle réellement modifiée après l'appel ?

14.1 Premier exemple

Voici un exemple écrit en Python :

```
# Procédure ajoute
def ajoute(a):
  a = a + 1

# Programme principal
b = 5
ajoute(b)
print(b)
```

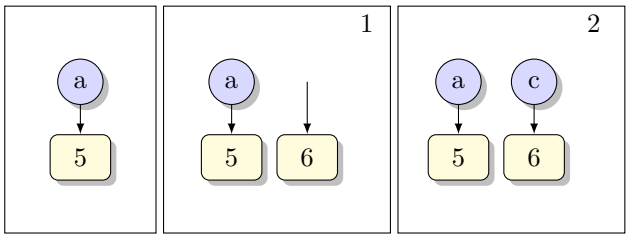
La question est de savoir ce que va afficher la ligne **`print(b)`**. 5 ou 6 ?
En Python, le passage des paramètres est comparable à une affectation. Le fil d'exécution ressemble donc à ceci :

```
b = 5
# exécution de ajoute(b) :
a = b
a = a + 1
# retour au prog principal
print(b)
```

Sur cet exemple, il est évident que le programme affichera 5 (c'est `a` qui contient 6, pas `b`).
Le mécanisme responsable de ce comportement est plus complexe qu'il n'y parait, et est assez spécifique à Python. *Tout se passe comme si le paramètre était passé par valeur* : une copie du contenu de la variable est recopié de `b` vers `a` et c'est la copie qui est modifiée, pas l'original.
Pour comprendre le mécanisme réel, détaillons les deux lignes de code suivantes :

```
a = 5
c = a + 1
```

Nous avons déjà mentionné que les variables, en Python, étaient des *références* vers des objets. Cette nuance est maintenant centrale. La figure suivante illustre la manière dont les lignes précédentes, et tout particulièrement la seconde, sont exécutées[8] :

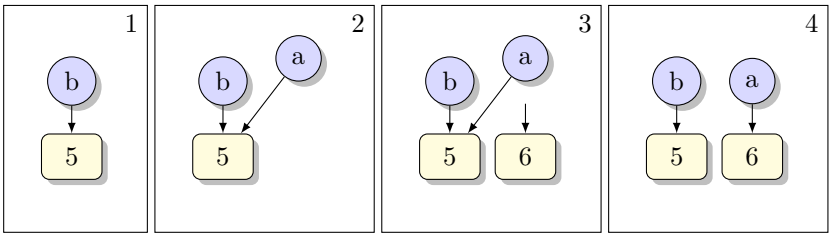


La ligne `c=a+1` est décomposée en :

1. L'objet à droite du symbole d'affectation est créé, sauf si c'est une référence simple (ici, création de l'objet 6).
2. La variable à gauche du symbole d'affectation référence maintenant le nouvel objet créé.

Pour les variables numériques, on peut tout ignorer de ce mécanisme, et considérer que les variables sont de simples «tiroirs» pour ranger les nombres. Mais connaître ce mécanisme permet d'expliquer les autres comportements de Python que nous allons détailler dans la suite.
Voyons à présent ce qui se produit lors du passage d'un paramètre :

```
b = 5
# exécution de ajoute(b) :
a = b
a = a + 1
```



1 `b=5` : on crée l'objet 5, et `b` est une nouvelle référence vers cet objet

2 `a=b` : on a une simple référence à droite du symbole `=`, `a` devient une nouvelle référence vers l'objet `b`, qui existait déjà.

3 et 4 `a=a+1` : on crée l'objet spécifié à droite du `=` (6), et `a` devient une référence vers ce nouvel objet.

On comprend donc pourquoi la valeur indiquée dans `b` n'a pas changé.

8. Dans cette figure, les objets sont représentés par des carrés ou des rectangles, et les références par des cercles. Nous conserverons cette convention par la suite.

14.2 Second exemple

Voyons maintenant ce qui se produit lorsqu'on passe une liste en paramètre :

```
# Procédure ajoute_liste
# ajoute v à la fin de la liste l
def ajoute_liste(l, v) :
  l.append(v)

# Programme principal
lst = [1, 2, 3]
ajoute_liste(lst, 42)
print(lst)
```

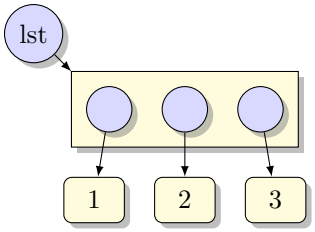
Nous posons toujours la même question : que va afficher le programme principal : `[1,2,3]` ou `[1,2,3,42]`.
Là aussi (ceci est toujours valable), le passage des paramètres est comparable à une affectation. Aussi, la machine exécute :

```
lst = [1, 2, 3]
# exécution de ajoute_liste(lst,42)
l = lst
v = 42
l.append(v)
# retour au prog principal
print(lst)
```

Cette fois-ci, c'est `[1,2,3,42]` qui va s'afficher, comme si la procédure `ajoute_liste` avait pu modifier la liste originale, de manière contradictoire avec ce qui se passait précédemment avec la procédure `ajoute` et un nombre entier...
Dans cet exemple, *tout se passe comme si la liste était passée par adresse*. C'est donc la liste originale qui est modifiée dans la procédure.
Examinons dans un premier temps ce qui se cache derrière la simple ligne :

```
lst = [1, 2, 3]
```

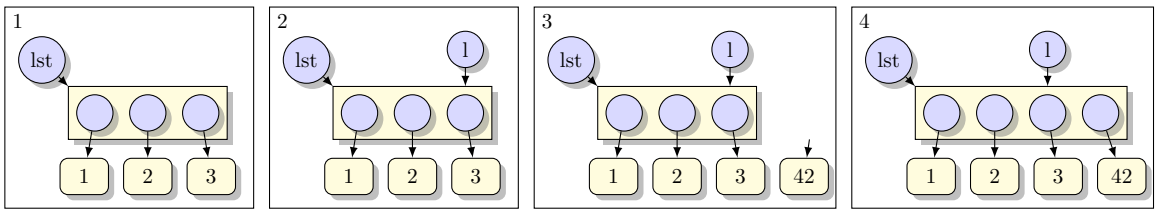
Lors de l'exécution de cette instruction, les objets 1, 2 et 3 sont créés, suivis d'une liste qui contient les références vers chacun des objets : les élément de la liste ne *sont pas* les objets mais *des références vers ces objets*. Cette liste est enfin référencée par `lst`. On se trouve dans cette situation :



Voyons maintenant ce qui se cache derrière le passage de paramètres de notre second exemple :

```
lst = [1, 2, 3]
# exécution de ajoute_liste(lst,42)
# on fait grâce du passage de 42...
l = lst
l.append(42)
```

Le déroulement des opérations est le suivant :



Voici ce qui se produit :

1 `lst=[1,2,3]` : la liste `lst` est créée, et contient des références vers 3 entiers.

2 `l=lst` : une nouvelle référence à la liste est ajoutée

3 et 4 `l.append(42)` : l'objet 42 est créé. Puis, la méthode `append` demande à la liste référencée par `l` de stocker une référence vers l'objet passé en paramètre.

Dans ce cas, la liste d'origine (celle qui était désignée par `lst` est bel et bien modifiée. Ceci explique que la liste du programme principal, qui contenait `[1,2,3]`, contient effectivement quatre éléments après l'appel à `ajoute_liste`.

14.3 Troisième exemple

Voici maintenant un troisième exemple qui va nous permettre de terminer sur le mécanisme de passage des paramètres en Python.

```
# Procédure ajoute_liste
# crée une liste contenant la liste d'origine
# plus un élément
def ajoute_liste2(l, v):
  l = l + [v]

# Programme principal
lst=[1, 2, 3]
ajoute_liste2(lst, 42)
print(lst)
```

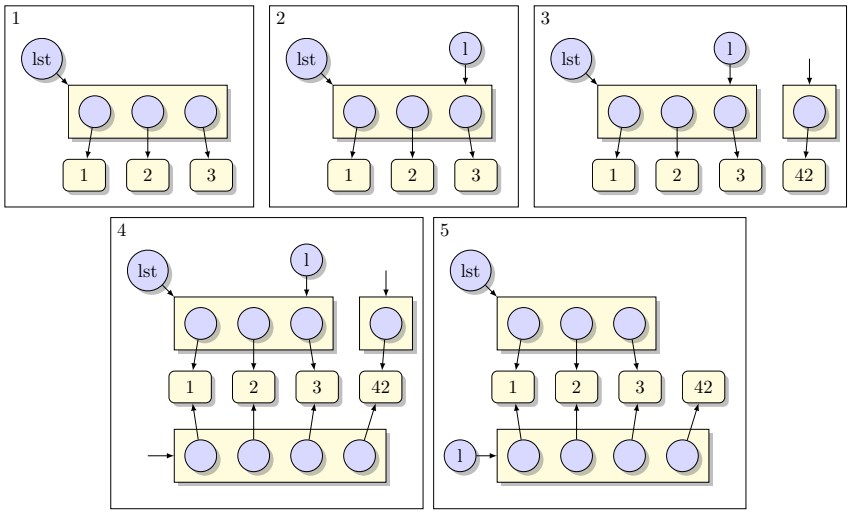
Le passage des paramètres est toujours équivalent à une affectation et tout se passe comme si nous exécutions :

```
lst = [1, 2, 3]
# exécution de ajoute_liste2(lst, 42)
l = lst
v = 42
l = l + [v]
# retour au prog principal
print(lst)
```

Cette fois-ci le programme affiche `[1,2,3]`...
Ceci peut paraître surprenant après la lecture du second exemple, mais s'explique très simplement si nous reprenons nos schémas d'objets et de références.
Dessinons les objets mis en jeu dans le code :

```
lst = [1, 2, 3]
# exécution de ajoute_liste2(lst, 42)
l = lst
l = l + [42]
```

Voici le déroulement des opérations :



L'exécution semble un peu plus complexe :

1. `lst=[1,2,3]` : la liste `lst` est créée, et contient des références vers 3 entiers.
2. `l=lst` : une nouvelle référence vers la même liste est ajoutée

La ligne suivante : `l=l+[42]` correspond, comme ça a été le cas jusqu'à présent, à la création de l'objet à droite du signe `=`. Puis `l` devient une référence vers ce nouvel objet.
La création de l'objet `l+[42]` est la création d'une *nouvelle liste*, concaténation de deux listes (`l` et `[42]` [9], de la même manière que `3+5` provoquerait la création d'un nouvel entier, somme de `3` et de `5`.

3. création de la liste temporaire `[42]`, afin de l'utiliser comme opérande.
4. création de la troisième liste, concaténation des deux autres. Notez que les objets (1,2, etc...) ne sont pas recopiés, ce sont les références vers ces objets (le contenu d'une liste, ce sont les références vers les objets) qui le sont.
5. La nouvelle liste s'appelle maintenant `l`. L'ancienne liste `lst` n'a donc pas été modifiée.

Résumé

Lorsqu'on passe des objets en paramètres d'une procédure, il est nécessaire de savoir si les modifications qu'on leur apporte dans la procédure altèrent l'objet d'origine. Les cas simples sont réglés facilement par habitude (nombres entiers, simples listes), mais lorsqu'on hésite (liste de listes par exemple), une solution est de revenir au modèle d'affectation de Python, qui permet dans tous les cas d'expliquer ce qui se produit.

15 Récursivité

recursivité *: n.f.*
1. Voir : récursivité

15.1 Récurrence en mathématiques

Nous allons commencer par l'exemple de la fonction factorielle, qui peut être définie de manière récursive :

$$\forall n \geqslant n! = \text{fact}(n) = \begin{cases} n \times \text{fact}(n-1) & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

Voici quelques éléments qui sont toujours présents dans une définition récursive :
- l'objet de taille n est défini à partir d'un objet *de taille plus petite* ($\text{fact}(n)$ est défini à partir de $\text{fact}(n-1)$). Ce point n'est pas toujours évident.
- les *petits objets* ne sont pas définis de manière récursive (c'est le cas de $\text{fact}(0)$ dans notre exemple).

15.2 Algorithme récursif

De la même manière qu'une définition peut être récursive, un algorithme peut être récursif.

```
fonction fact(n: entier) : entier
    si n>0 :
        r←n*fact(n−1)
    sinon
        r←1

    retourner r
```

```
def fact(n):
  if n > 0:
    r = n * fact(n - 1)
  else:
    r = 1
  return r
```

Comment un algorithme récursif fonctionne-t-il ?

Une fonction récursive utilise la pile des appels pour stocker les résultats intermédiaires. Il n'est pas *nécessaire* de savoir ça pour écrire des fonctions récursives.

9. C'est pour que `[42]` soit bien une liste et pas un nombre qu'il y a des crochets.

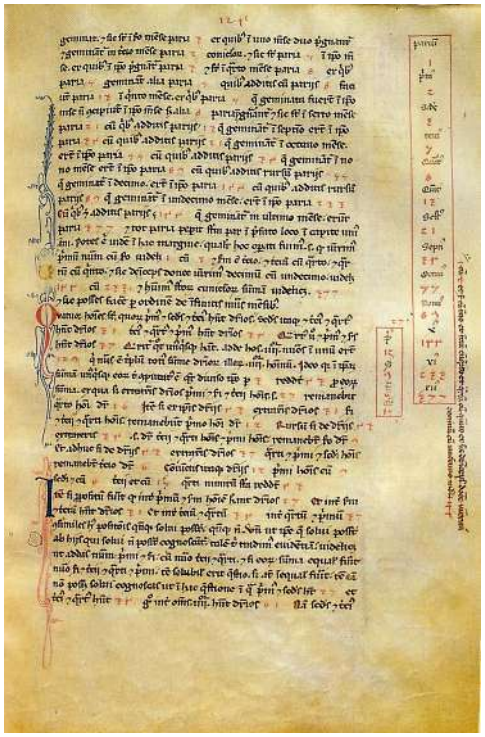
15.3 Quelques exemples

Suite de Fibonacci

Wikimedia, Fibonacci2.jpg

La suite de Fibonacci est définie ainsi :

$$\begin{cases} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \text{ si } n > 1 \end{cases}$$



Wikimedia, Liber_abbaci_magliab_f124r.jpg

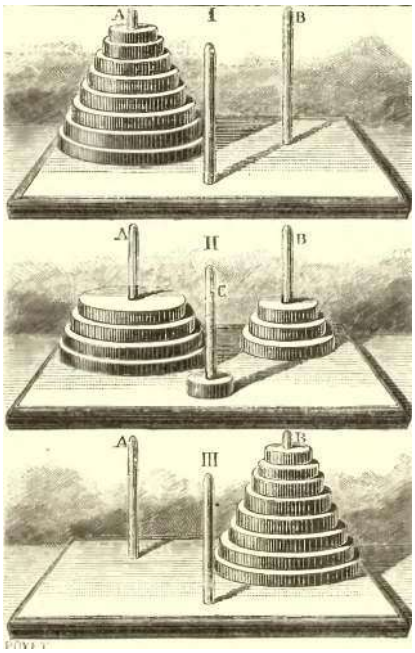
```
def fibo(n):
  if n < 2:
    return n
  return fibo(n - 1) + fibo(n - 2)
```

Les tours de Hanoï

Le jeu consiste à déplacer une tour de plusieurs disques d'un piquet vers un autre, en déplaçant les disques un par un, sans jamais poser un grand disque sur un plus petit.

Extrait de l'*Arithmétique amusante* de Lucas ($\approx$ 1895) :

> Le mandarin N. Claus (de Siam) [10] nous raconte qu'il a vu, dans ses voyages pour la publication des écrits de l'illustre Fer-Fer-Tam-Tam, dans le grand temple de Bénarès, au-dessous du dôme qui marque le centre du monde, trois aiguilles de diamant, plantées dans une dalle d'airain, hautes d'une coudée et grosses comme le corps d'une abeille. Sur une de ces aiguilles, Dieu enfila au commencement des siècles, 64 disques d'or pur, le plus large reposant sur l'airain, et les autres, de plus en plus étroits, superposés jusqu'au sommet ; c'est la tour sacrée de Brahma. Nuit et jour, les prêtres se succèdent sur les marches de l'autel, occupés à transporter la tour de la première aiguille de diamant sur la troisième, sans s'écarter des règles fixes que nous venons d'indiquer, et qui ont été posées par Brahma. Quand tout sera fini, la tour et les brahmes tomberont, et ce sera la fin des mondes !



archive.org, larithmetiqueam00lucarich

Il est possible d'écrire un algorithme récursif, très concis, qui indique la liste des déplacements à réaliser pour un nombre quelconque de disques. Exemple pour 3 disques : $1 \to 3,\ 1 \to 2,\ 3 \to 2,\ 1 \to 3,\ 2 \to 1,\ 2 \to 3,\ 1 \to 3$.
Cet exercice sera corrigé en cours...

10. du collège Li-Sou-Stian

16 Résoudre des problèmes : méthode de travail

Un algorithme présente généralement plusieurs étapes de calcul, qui sont plus ou moins immédiates.
Il faut être capable de :

- hiérarchiser les étapes de calcul
- découper le problèmes en sous-problèmes plus simples, et indépendants

On réappliquera la méthode à chaque sous-problème, jusqu'à arriver à des problèmes connus ou triviaux.
Les avantages de la décomposition en sous-problèmes sont nombreux et d'une grande importance :

- clarté du code ;
- répartition possible du travail entre plusieurs personnes ;
- possibilité de remplacer facilement un sous-algorithme par un autre plus efficace sans pour autant apporter de retouche à l'ensemble du programme.

16.1 Développement selon John Zelle

Dans son ouvrage *Python Programming : an introduction to computer science*[11], John Zelle donne les recommandations suivantes :

Analysez le problème Comprenez le problème à résoudre et sa nature. Essayez d'en apprendre le plus possible sur lui. Vous ne pourrez pas le résoudre avant de le connaître parfaitement.

Spécifiez le programme Décrivez très exactement ce que votre programme va faire. Vous ne devez pas déjà vous soucier de la façon dont il va le faire, mais plutôt décider très exactement ce qu'il va faire. Pour des programmes simples, cela consistera à décrire ce que seront les entrées et les sorties du programme et ce qui les relie.

Concevez le programme Formulez la structure globale du programme. C'est à ce moment que vous traiterez de la façon dont le programme résoudra le problème. Le travail principal est de concevoir le ou les algorithmes qui rempliront les tâches préalablement spécifiées.

Codez Traduisez les algorithmes conçus dans un langage de programmation et entrez les sur un ordinateur. Dans ce livre, nous programmerons nos algorithmes en Python.

Testez/Débuggez le programme Essayez votre programme et voyez s'il fonctionne comme vous le souhaitiez. S'il y a des erreurs (souvent appelées bugs), revenez à l'étape précédente et corrigez les. L'activité qui consiste à débusquer et corriger les erreurs s'appelle le debuggage[12]. Durant cette phase, votre objectif est de trouver des erreurs, et pour cela, vous devez tester tout ce qui vous passe par le tête et qui pourrait planter le programme. Il sera utile de garder à l'esprit la maxime : *Nothing is foolproof because fools are too ingenious*[13]

Faites de la maintenance Continuez à développer votre programme en répondant aux besoins des utilisateurs. La plupart des programmes ne sont jamais vraiment finis ; ils continuent d'évoluer au fil des années d'utilisation.

16.2 Rechercher un élément dans une liste triée

Nous allons nous intéresser au problème suivant : Nous disposons d'une liste d'éléments classés par ordre croissant. Nous voulons vérifier si une valeur donnée est dans la liste ou pas. Ce petit problème peut se rencontrer à l'intérieur de nombreux autres problèmes.

Une première solution

Voici un exemple de liste sur laquelle nous pourrions être amenés à travailler :

-5	-2	3	6	7	9

La première étape consiste à avoir une idée sur la façon de procéder. Le problème n'étant pas très difficile, nous pouvons envisager une première solution :

> *Commencer sur la première case. Tant que la case examinée est strictement inférieure au contenu recherché, avancer d'une case. Si on épuise la liste ou qu'on tombe sur une valeur strictement supérieure à la valeur recherchée, c'est qu'elle n'est pas dans la liste. Sinon, elle y est.*

11. Deuxième édition, Franklin, Beedle & Associates
12. On dit parfois déverminage en français, mais personnellement, je trouve que ça supprime le côté affectif... ;-)
13. Rien n'est à l'épreuve des imbéciles, car ils sont trop malins.

Nous devons maintenant essayer de formaliser un peu plus notre algorithme.
En premier lieu, nous allons choisir comme structure de données, une liste (ou un tableau) telle qu'on en trouve dans Python. La numérotation des cases commencera à 0 [14]. L'important pour la suite est que la structure de données doit pouvoir stocker une collection d'objets, ordonnés, et numérotés.
Cette étape de formalisation est difficile à franchir lorsqu'on débute en programmation :

```
fonction recherche(l : liste , v: valeur)
    i←0
    repeter tant que i<longueur(l) et l[i]<v
    |  i←i+1

    si  i≥longueur(l) ou l[i]>v :
    |  retourner Faux
    sinon
    |  retourner vrai
```

Il ne nous reste plus qu'à traduire cet algorithme, par exemple en Python :

```
def rechercher(l, v):
    i = 0
    while i < len(l) and l[i] < v:
        i = i + 1
    if i >= len(l) or l[i] > v:
        return False
    else:
        return True
```

De la même idée de départ nous aurions pu décliner un algorithme formel un peu différent et finir sur un autre programme tout aussi correct :

```
def rechercher(l, v):
    i = 0
    while True:
        if i >= len(l) or l[i] > v:
            return False
        if l[i] == v: return True
        i = i + 1
```

Plusieurs méthodes pour un problème ?

Notre méthode (notre algorithme) est-elle efficace ?

- Si la valeur cherchée est plus grande que la dernière case, nous devons tout parcourir
- Cette méthode revient à rechercher un mot dans le dictionnaire en tournant les pages **une par une** depuis la première jusqu'à la bonne page...

Reprenons la comparaison avec le dictionnaire pour essayer de dégager une autre méthode : Pour rechercher dans un dictionnaire, nous tournons les pages par paquets, quitte à parfois revenir en arrière. Même si la recherche dans un dictionnaire, manuellement, est approximative, l'idée est d'ouvrir le dictionnaire, et d'en déduire si le mot à chercher est avant ou après. Une fois ceci décidé, on ouvre à nouveau le dictionnaire sur la portion sélectionnée et on compare à nouveau avec le mot cherché. À chaque étape, nous réduisons ainsi à peu près de moitié le nombre de pages dans lesquelles il reste à effectuer la recherche.

Algorithme de la méthode dichotomique Pour chercher v entre les cases g et d, regarder le contenu de la case du milieu m ($m = (g + d)/2$). S'il est plus petit que v, continuer à chercher dans l'intervalle m, d sinon, continuer à chercher dans l'intervalle g, m.
Voici l'algorithme, écrit de manière plus formelle :

```
Pour rechercher v dans la liste l :
    g,d ←0,len(l)−1
    répéter tant que la «tranche» g,d fait plus d'une case :
    |  m←milieu de g,d
    |  si v>l[m] : g←m+1
```

14. la numérotation à partir de 0 est habituelle en informatique, même si elle n'est pas utilisée dans *tous* les langages.

```
    | sinon : d←m
    si l[g]=v : retourner Vrai
    sinon retourner Faux
```

Attention aux difficultés cachées

Les raccourcis, qui n'existeront plus dans le programme final (tout sera détaillé) peuvent contenir des erreurs potentielles. Nous avons par exemple écrit que m était le milieu de g, d. Tous ces nombres étant entiers, la manière de faire l'arrondi sur m a son importance.

Voici le programme Python correspondant :

```
def cherchedicho(l, v):
    g, d = 0, len(l) - 1
    while g != d:
        m = (g + d) // 2
        if v > l[m]: g = m + 1
        else : d = m
    if l[g] == v: return True
    else: return False
```

Cette fonction a beau être courte et assez simple, elle peut contenir des erreurs qui ne sont pas évidentes à détecter. Par exemple, si nous avions écrit :

```
# ATTENTION, cette fonction ne marche pas
def cherchedicho(l, v) :
    g, d = 0, len(l) - 1
    while g != d:
        m= (g + d) // 2
        if v < l[m]: d = m - 1
        else: g = m
    if l[g] == v: return True
    else: return False
```

alors le programme n'aurait pas fonctionné. Vous vous en rendrez compte en exécutant à la main ce que fait l'algorithme pour rechercher la valeur 5 dans la liste `[3,4]`.

Exécution manuelle

Exécuter un algorithme à la main, pour quelques jeux de valeur, est essentiel dans la détection des erreurs. Il faut le faire de manière systématique, et ne pas s'en priver, à plus forte raison, lorsque l'algorithme est simple.

Comparaison des méthodes

Nous voyons donc que pour résoudre un problème particulier, il y a plusieurs méthodes. Une fois la méthode (l'algorithme) choisie, on peut encore trouver des programmes avec des petites variantes.
Est-il intéressant de trouver plusieurs méthodes ou la première fera-t-elle tout le temps l'affaire ?
La figure II.2 (gauche) représente en abscisse le nombre de valeurs de la liste (la valeur max est environ 10 millions) et en ordonnée le temps mis pour rechercher 200 valeurs dans cette liste (valeur max : 0.1 secondes). Cette courbe a été obtenue en utilisant la méthode dichotomique.
La figure II.2 (droite) représente [15] les temps d'exécution de la première méthode, pour les mêmes essais, ainsi que, plaqué sur l'axe des abscisses, à nouveau les résultats de la méthode dichotomique.

Notion de complexité

Pouvions nous prédire les performances de ces algorithmes sans les programmer, afin d'estimer leurs mérites respectifs et choisir la bonne méthode ?
Dans le cas de la recherche naïve, le nombre d'opérations dans la première boucle est proportionnel à la position de la valeur dans le tableau. La fin de l'algorithme s'exécute en temps constant. En moyenne, le nombre d'opérations est donc $k_1 \frac{n}{2} + k_2$. On dit que l'algorithme est en $\Theta(n)$ ou linéaire. Cela signifie que son temps d'exécution varie comme n. Autrement dit, lorsque n est grand, si on multiplie la taille des données par k, le temps d'exécution sera lui aussi multiplié par k.

15. En rouge si vous avez la couleur...

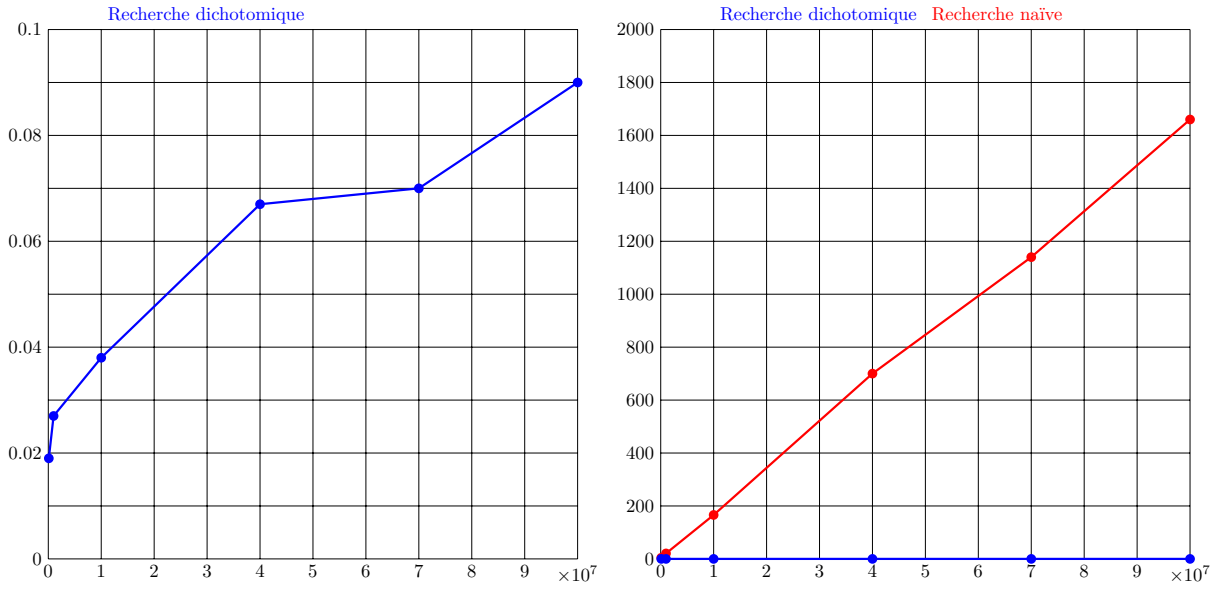


Figure II.2 – Comparaison des temps (s) d'exécution de deux algorithmes de recherche

Notation de Landau

Une fonction $g(n)$ appartient $\Theta(f(n))$ si :

$$\exists n_0, k_1 > 0, k_2 > 0 | \forall n > n_0, k_1 f(n) \leqslant g(n) \leqslant k_2 f(n)$$

La notation Θ permet en quelques sorte d'encadrer le comportement asymptotique d'une fonction.
Une fonction $g(n)$ appartient $O(f(n))$ si :

$$\exists n_0, k > 0 | \forall n > n_0, g(n) \leqslant k f(n)$$

La notation O permet donc de majorer le comportement asymptotique d'une fonction.

Malgré cette différence, on trouve très souvent, dans les calculs de complexité, O à la place de Θ. Il convient donc de faire attention à la convention utilisée par l'auteur.

Dans le cas de la recherche dichotomique, le corps de la boucle s'exécute en temps constant. La boucle s'exécute jusqu'à ce que g et d soient égaux. La distance $g - d$ est divisée par 2 à chaque tour. On a donc $\ln_2(n)$ tours de boucle. Le nombre d'opérations est donc : $k_1 \ln_2(n) + k_2$. On dit que l'algorithme est en $\Theta(log(n))$ ou logarithmique. Autrement dit lorsque n est grand, pour doubler le temps d'exécution, il faut élever le nombre de cases au carré ! Cet algorithme est donc incroyablement plus rapide que le précédent.

Est-ce vraiment important ?

Un algorithme logarithmique est **considérablement** plus efficace qu'un algorithme linéaire (lui-même considérablement plus efficace qu'un algorithme quadratique, ou encore exponentiel).

Cette différence devient fondamentale si la taille des données est importante.

On peut estimer que pour un tableau de 10^{10} valeurs,

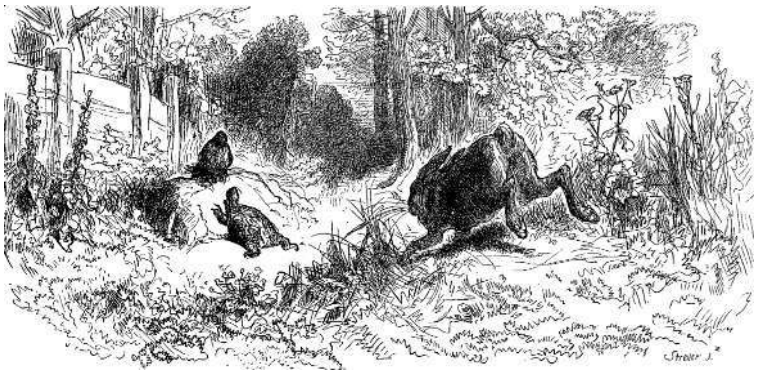
1. le temps moyen pour retrouver un seul nombre serait d'environ *2 heures* dans le premier cas
2. il serait de moins d'une *milliseconde* dans l'autre cas

Calcul de complexité

Pour détermine le coût $c(a)$ d'un algorithme a, on peut faire le décompte suivant :

- les opérations «atomiques» : affectation, comparaison, opérations arithmétiques ou composition de tout ça (sans boucle) sont considérées comme ayant un temps d'exécution unitaire (ou tout au moins constant).
- Coût d'une séquence `p; q` : $c(\'p; q\check{}) = c(p) + c(q)$
- Coût du test `if b: p else: q` : borné par $c = c(b) + max(c(p), c(q))$
- Coût d'une boucle `for i in it: p`
  - $c = n \times c(p)$ avec $n =$ `len`(it) si $c(p)$ ne dépend pas de i
  - $c = \sum_{i \in it} c(p_i)$ sinon

— Coût d'une boucle `while b: p` somme des $c(b)+c(p)$ à chaque tour de boucle (on peut essayer de majorer....)



Le lièvre et la tortue – Gustave Doré

16.3 Algorithmes de tri

Nous allons nous intéresser maintenant à un second problème, celui des très classiques algorithmes de tri.
Nous supposons que nous disposons d'une liste d'objets, par exemple :

-2	9	-5	7	3	6

Nous devons écrire un algorithme capable de fournir en retour une liste triée par ordre croissant :

-5	-2	3	6	7	9

Tri par sélection

Une première idée est de procéder ainsi :

1. rechercher le plus petit, le mettre au début,
2. rechercher le second plus petit, le mettre en deuxième position
3. ...

Une fois le minimum placé en première position, tout se passe comme si on recommençait la même chose sur tout le tableau privé du premier élément. Puis à nouveau sur tout le tableau privé des deux premiers etc...
Voici l'algorithme correspondant. Notez comment le problème a été décomposé en deux parties : l'algorithme général `tri_selection` et la fonction `rechmin`, qui recherche le plus petit élément dans une portion de tableau. En cas, de problème, nous pouvons tester séparément la fonction `rechmin` et ainsi cibler plus facilement les erreurs.

```
fonction tri_selection (tableau t)
    /* Trie le tableau **sur place** */
    n ← taille (t)
    repeter pour i de 0 à n−2 :
        j←rechmin(t,i)
        échanger le contenu des cases i et j

fonction rechmin(tableau t, entier i) : entier
    min=i
    repeter pour j de i+1 à taille (t)−1 :
        si t[j]<t[min] : min←j

    retourner min
```

Cet algorithme est quadratique (complexité en $\Theta(n^2)$).

Tri rapide

Voici une autre idée d'algorithme (probablement moins facile à avoir) :

1. Sélectionner une valeur au hasard dans le tableau : le pivot
2. Organiser le tableau en deux parties : à gauche les éléments inférieurs au pivot et à droite, les éléments supérieurs au pivot.
3. Trier les deux parties avec la même méthode

Partitionner le tableau

Il existe plusieurs façons de partitionner le tableau, qui sont plus ou moins équivalentes, mais l'idée reste la même : mettre les petits éléments à gauche, et les grands à droite.

Voici l'algorithme du tri rapide (version CLR) :

```
fonction tri_rapide (tableau t, indices d,f)
    si f>d :
        p←partitionner(t,d,f)
        tri_rapide (t,d,p−1)
        tri_rapide (t,p+1,f)

fonction partitionner (tableau t, indices d,f) : entier
    p←t[f] /* le pivot */
    i ←d − 1
    repeter pour j de d à r − 1:
        si t[j] ≤p:
            i ←i + 1
            echanger t[i] et t[j]

    échanger t[f] et t[i + 1]
    retourner i + 1
```

— La fonction `partitionner` a un temps d'exécution proportionnel à la taille de la tranche de tableau.
— La fonction `tri_rapide` est récursive. Son temps d'exécution est donné par une relation de récurrence :

$$T(n) = \overbrace{a\,n}^{\text{partitionner}} + 2 \times \underbrace{T(n/2)}_{\substack{\text{puisqu'en moyenne, p est} \\ \text{au milieu de la tranche}}}$$

Tous calculs faits, on trouve une complexité en $\Theta(n \log(n))$
Comme pour le cas de la recherche, connaître la complexité des algorithmes est une indication particulièrement utile dès que l'on a beaucoup de grandes quantités de données. La figure II.3 montre les temps mis pour trier des tableaux de 500 à 10 millions de valeurs, en utilisant 3 algorithmes différents : le tri par fusion, le tri par sélection et le tri rapide.

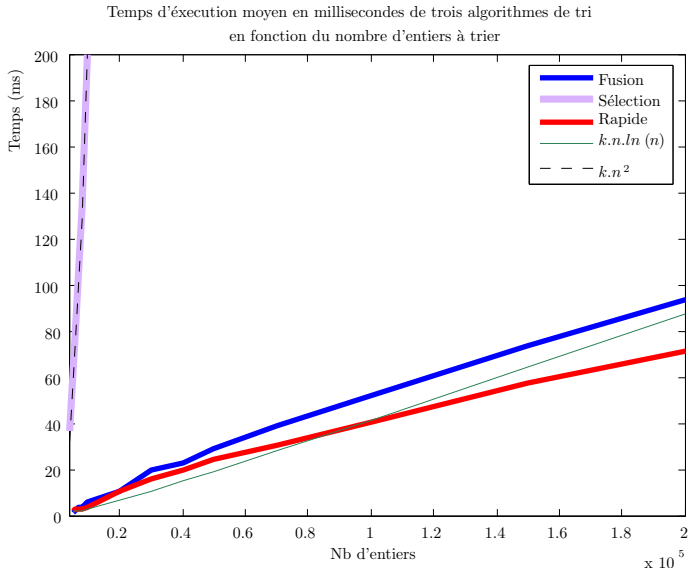


Figure II.3 – Comparaison des temps d'exécution de plusieurs algorithmes de tri

— pour $n < 500$ le temps d'exécution reste inférieur à 1 milliseconde.
— pour $n = 100\,000$, le tri par sélection met quelques dizaines de secondes à s'exécuter, contre 0,4 secondes pour le tri rapide.
— pour $n = 10$ millions (estimation), le tri par sélection mettrait plus de 2 jours, contre environ 5 secondes pour le tri rapide.

17 Récursivité et Logo

Le monde tel qu'il est vu par Logo, un langage à but pédagogique inventé dans les années 60 par Seymour Papert, est un espace en deux dimensions [16] dans lequel se déplace une tortue, qui peut ou non laisser une trace des endroits où elle passe. Un programme Logo est composé d'ordres donnés à la tortue (comme «avance de 10» ou «tourne à gauche de 90 degrés»).
Nous allons ici utiliser le module `turtle` de Python, qui permet de programmer *comme en Logo*. Les programmes pouvant être récursifs, c'est l'aspect graphique du Logo et la possibilité d'utiliser la récursivité qui permet de réaliser des figures complexes en très peu de lignes.

17.1 Écriture des primitives

Le programme suivant trace un carré ayant pour côté la valeur passée en paramètre :

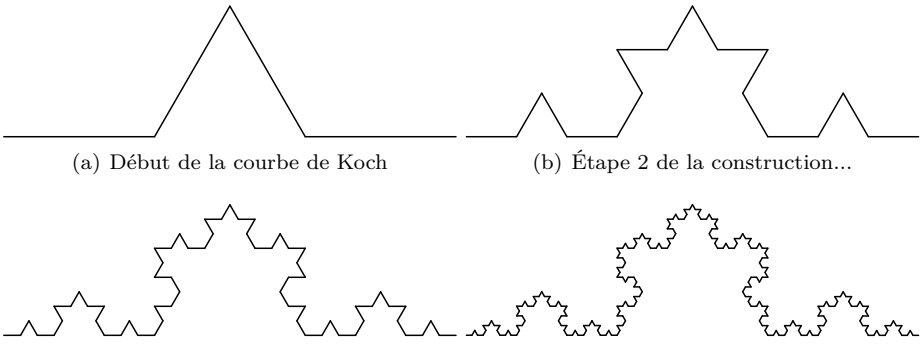
```
def carre(cote):
 for i in range(4):
    fd(cote)
    rt(90)
```

Pour utiliser ce programme, il faut ensuite entrer, dans un shell Python :

```
from turtle import *
carre(100)
```

17.2 Courbe de Koch

Supposons que nous désirions créer la figure II.4(a), chaque segment étant de longueur égale et quelconque.



(a) Début de la courbe de Koch (b) Étape 2 de la construction... (c) Étape 3 de la construction... (d) Courbe de Koch

FIGURE II.4 – Construction de la courbe de Koch

Nous écririons la primitive suivante :

```
def  fig(l):
   fd(l)
   lt(60)
   fd(l)
   rt(120)
   fd(l)
   lt(60)
   fd(l)
```

Considérons à présent la figure II.4(b).
C'est la même que précédemment, excepté que chaque segment a été remplacé par un modèle réduit de la figure elle même.... Sur la figure II.4(c), nous avons à nouveau remplacé chaque segment par le dessin de départ. Et ainsi de suite, jusqu'à obtenir (à l'infini...), la courbe complète (figure II.4(d)).
Naturellement, l'infini est ici lié à la taille d'un pixel à l'écran.

16. À l'origine il n'y avait que deux dimensions. Les versions actuelles de Logo permettent de faire de la 3D.

Comment tracer de telles figures ? La réponse est très simple, et correspond tout à fait à la façon de tracer la figure à la main. Il suffit de remplacer chaque segment par la figure elle-même, trois fois plus petite, comme dans le programme suivant :

```
# Attention, cette version n'est pas fonctionnelle
def fig(l):
    fig(l / 3)
    lt(60)
    fig(l / 3)
    rt(120)
    fig(l / 3)
    lt(60)
    fig(l / 3)
```

Ce programme comporte néanmoins une grosse erreur. Lorsque `longueur` vaudra par exemple 1, la tortue va continuer la *descente récursive* et vouloir tracer la figure pour une arête de $\frac{1}{3}$, puis $\frac{1}{9}$, ... sans jamais s'arrêter... Nous devons donc, de même que nous arrêtons la descente récursive lors d'un dessin à la main, indiquer à la tortue à quel moment elle ne doit plus remplacer un segment potentiel par la figure complète, mais bien tracer le segment. Nous pouvons par exemple décider que lorsque la longueur sera inférieure ou égale à un, alors, il faudra arrêter la descente récursive. À ce moment, au lieu de tracer le dessin de la figure II.5(a), il faudra faire un simple segment comme indiqué figure II.5(b).

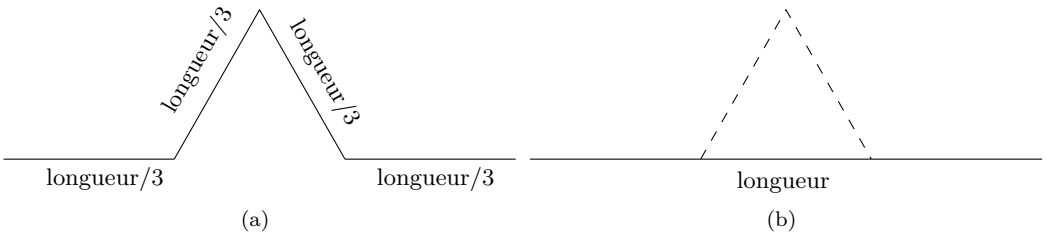


Figure II.5 – Arrêt de la récursivité pour la courbe de Koch

Voici maintenant notre programme complet et fonctionnel :

```
def fig(l):
    if l < 2 :
        fd(l)
        return
    fig(l / 3)
    lt(60)
    fig(l / 3)
    rt(120)
    fig(l / 3)
    lt(60)
    fig(l / 3)
```

17.3 Autres figures

Il existe de nombreuses courbes, que l'on peut obtenir simplement de manière récursive. En voici quelques-unes :

Triangle de Sierpinski Le triangle de Sierpinski est obtenu par les opérations suivantes. On dessine tout d'abord un triangle (figure II.6(a)), puis, on divise en 4 triangles (figure II.6(b)). Chaque triangle extérieur (il y en a 3) est considéré comme le triangle de départ. Il est donc divisé en 4 (figure II.6(c)). Au bout de quelques itérations, on obtient la figure II.6(d).

Courbe du dragon La récursivité est parfois «croisée» en ce sens que ce n'est pas toujours la procédure A qui s'appelle elle-même. Ce peut être la procédure A qui appelle la procédure B, et la procédure B qui appelle la procédure A. C'est par exemple le cas dans la courbe du dragon. Sur la figure II.7. sont représentées les courbes :

1. dragon droit de profondeur 1

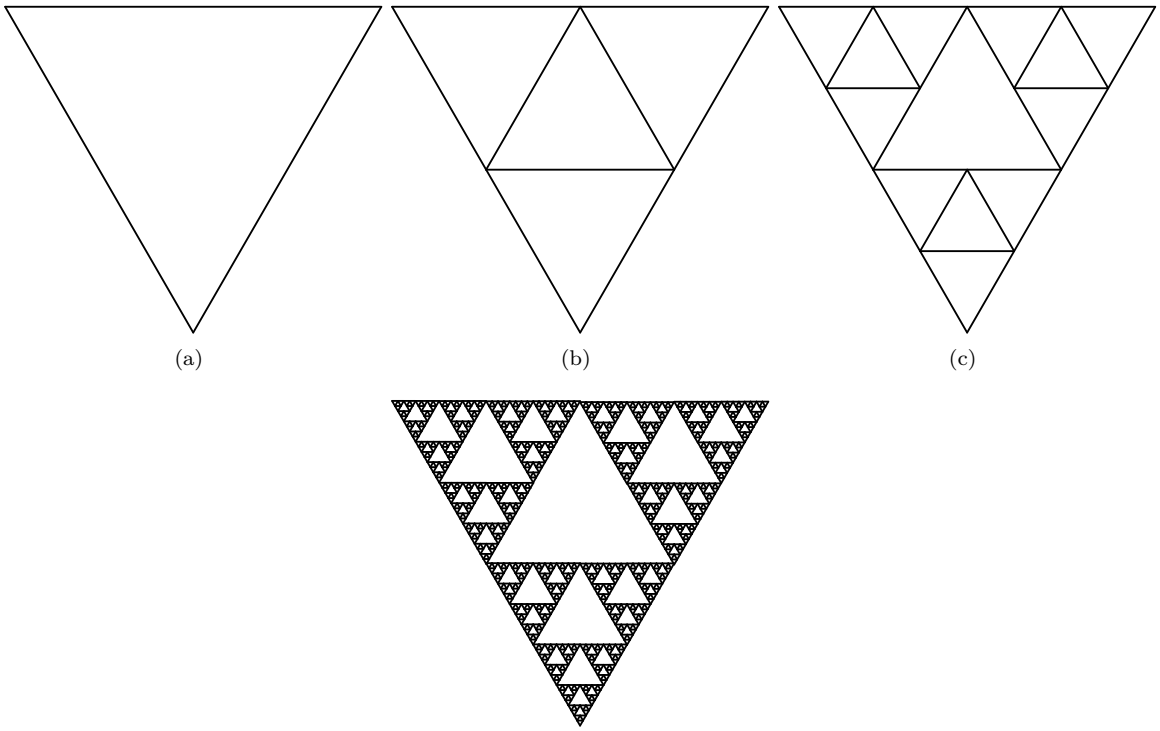


FIGURE II.6 – Tracé du triangle de Sierpinski

2. dragon gauche de profondeur 1
3. dragon droit de profondeur 2
4. dragon gauche de profondeur 2
5. dragon droit de profondeur 3
6. dragon gauche de profondeur 3
7. dragon gauche de profondeur 5
8. dragon gauche de profondeur 10 (les cases sont sont plus petites que précédemment)

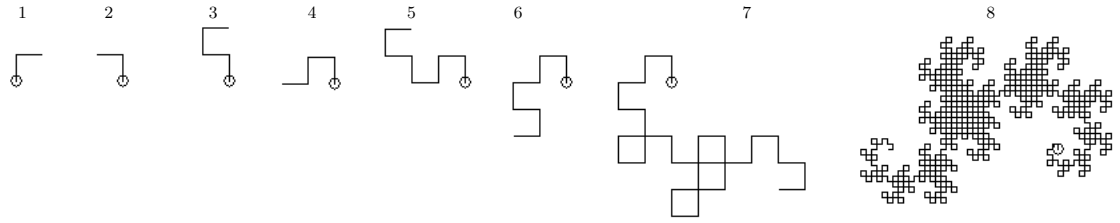


FIGURE II.7 – Courbe du dragon

Vous devriez remarquer (mais ce n'est pas facile) qu'un dragon gauche de profondeur n est composé d'un dragon gauche de profondeur $n-1$ d'une rotation de $+\frac{\pi}{2}$, et d'un dragon droit de profondeur $n-1$. Inversement un dragon droit de profondeur n est composé d'un d'un dragon gauche de profondeur $n-1$ d'une rotation de $-\frac{\pi}{2}$, et d'un dragon droit de profondeur $n-1$. Enfin, un dragon de profondeur 0 est simplement un trait.
Si l'on réduit la taille des traits à 1, on obtient comme dragon gauche de profondeur 15 l'objet de la figure II.8.

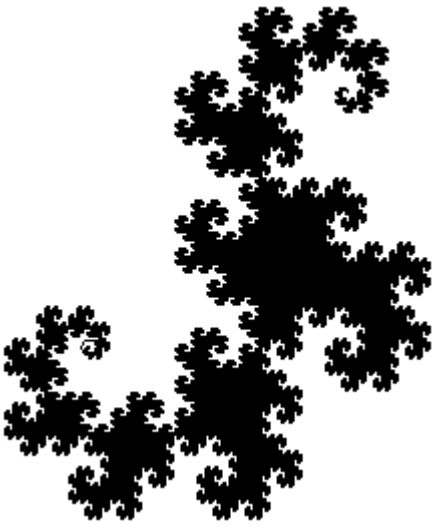
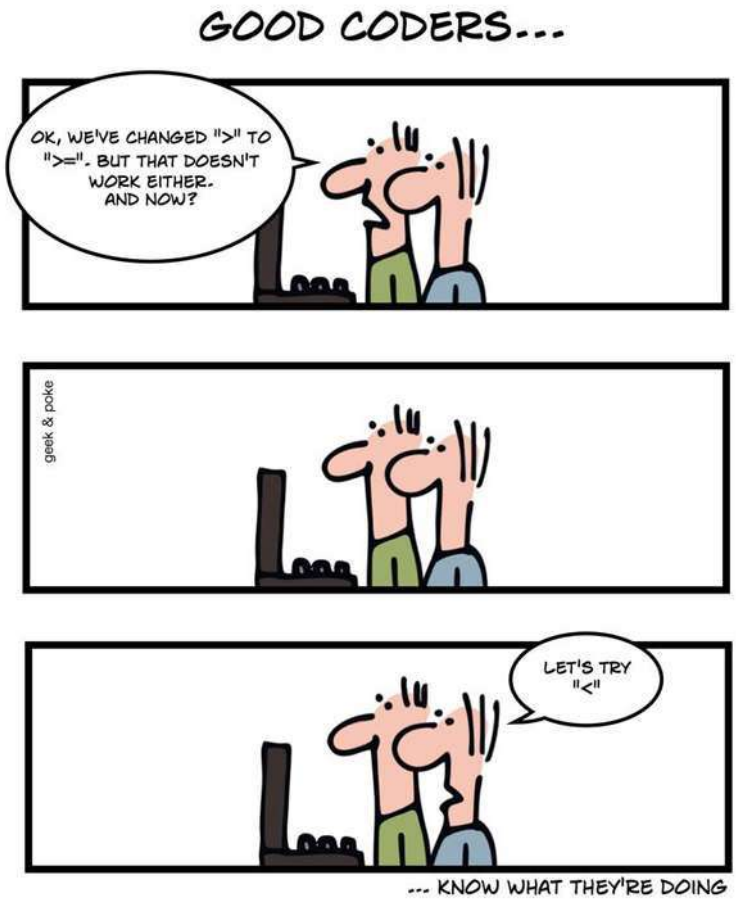


Figure II.8 – Dragon gauche de profondeur 15

Chapitre III

Compléments sur Python



Geek & Poke : http://geek-and-poke.com

Dans ce chapitre sont regroupées des informations sur l'utilisation de Python. Elles ne sont pas exhaustives (le lecteur est encouragé à consulter la documentation officielle), mais pourront aider à débloquer rapidement bon nombre de situations.

18 Types simples

18.1 Entiers

Le type **int**, permet de représenter les entiers de taille machine (quelques octets) ou les grands entiers (limités uniquement par la taille de la mémoire). Le passage de l'un à l'autre (entiers machines ou grands entiers) est transparent pour l'utilisateur depuis la version 3 de Python.
Pour indiquer une valeur entière, il est possible d'utiliser différentes bases :

```
>>> 42  # en décimal
```

```
>>> 0o52 # en octal
>>> 0x2A # en hexadécimal
>>> 0b101010 # en binaire
```

Les opérations disponibles sur les nombre entiers sont résumées dans la table III.1.

Opération	Type retour	Description
`x + y`	int	Somme
`x - y`	int	Différence
`x * y`	int	Produit
`x / y`	float	Quotient
`x // y`	int	Quotient entier
`x%y`	int	Reste de la division entière
`-x`	int	Opposé de `x`
`x&y`	int	Opération *et* sur les bits de `x` et `y`
`x | y`	int	Opération *ou* sur les bits de `x` et `y`
`x ^ y`	int	Opération *ou exclusif* sur les bits de `x` et `y`
`x << y`	int	Décalage à gauche de `y` bits sur `x`
`x >> y`	int	Décalage à droite de `y` bits sur `x`
`~x`	int	Inversion des bits de `x`
`abs``(x)`	int	Valeur absolue
`divmod``(x, y)`	(q,r)	Quotient (q) et reste (r) de la division entière de `x` par `y`
`float``(x)`	float	Conversion vers un float
`hex``(x)`	str	Représentation hexadécimale
`oct``(x)`	str	Représentation octale
`bin``(x)`	str	Représentation binaire
`int``(x)`	int	Conversion vers un int (pas de changement donc...)
`pow``(x, y[, z])`	int	`x` à la puissance `y` modulo `z` si `z` est précisé
`repr``(x)`	str	Représentation *officielle* sous forme de chaîne de caractères
`str``(x)`	str	Conversion en chaîne de caractères

TABLE III.1 – Opérations disponibles sur les nombre entiers

Les opérations de conversion vers les différentes bases sont réalisés avec les fonctions **`bin`**, **`oct`**, et **`hex`**.
Les opérations arithmétiques ordinaires sont disponibles sur les entiers :

```
>>> 6 + 9
15
>>> 6 - 9
-3
>>> 6 * 9
54
>>> 9 / 6
1.5
>>> 9 // 6 # division entière
1
>>> 9 % 6 # reste
3
```

Division entière

Notons qu'en Python 3, l'opérateur de division `/` est un opérateur de division non entière contrairement à ce qui se faisait en Python 2 et contrairement ce qui se fait dans de nombreux langages. Même si une division tombe juste, le résultat de l'opération `/` sera un **`float`**. Il faut donc penser, lorsqu'on souhaite manipuler des entiers, à utiliser l'opérateur `//`.

18.2 Booléens

Les deux seules valeur du type **`bool`** sont `True` et `False`. Ces deux valeurs peuvent être entrées litéralement ou peuvent être le résultat d'opérateurs de comparaison, comme `>`, `<`, `>=`, `<=`, `==` (égal), `!=` (différent).
Les opérateurs dont les opérandes sont des booléens sont les opérateurs logiques **`or`**, **`and`**, **`not`**.

Évaluation incomplète des expressions logiques

L'évaluation des expressions logiques n'est pas complète. *Elle s'arrête dès que le résultat est connu.* En logique, dans l'expression `a and b`, il suffit que `a` soit évalué à `False` pour que le résultat soit `False`. En Python, si `a` est évalué à `False`, le résultat de `a and b` sera `a`. De même si `a` est évalué à `True`, le résultat de `a and b` sera `b`. Ceci permet de construire des expressions riches, mais qui ne sont pas toujours très simples à comprendre. À notre niveau, il convient d'éviter de les écrire, mais il faut être capable de les comprendre.

Par exemple, `6<10 and 6*9` vaut 54 puisque `6<10` est vrai. Inversement, `42==6*7 or 6*9` vaut `True` (valeur de `42==6*7`)

En résumé, `x and y` :
- vaut `x` si `x` est faux (et alors `y` n'est pas évalué)
- vaut `y` si `x` est vrai (`x` et `y` sont évalués)

De même, `x or y` :
- vaut `y` si `x` est faux (`x` et `y` sont évalués)
- vaut `x` si `x` est vrai (et alors `y` n'est pas évalué)

Les expression qui ne sont ni le résultat de connecteurs logiques, ni le résultat d'opérateurs de comparaison sont évaluées à `True` sauf : `False`, `None`, la valeur `0` ou `0.0`, les collections vides (tuples, dictionnaires, listes, ensembles).

18.3 Nombres à virgule flottante

Le type **`float`** permet d'utiliser la représentation en virgule flottante (Norme ieee-754). Toute valeur numérique comportant un point décimal ou entrée en notation scientifique sera de type **`float`**. Le type **`float`** de Python correspond au codage en double précision [1] (c'est à dire au type double du C) :

```
>>> a = 7.54
>>> b = 7e3
>>> c = 0.745e1
```

Le module `math` contient de nombreuses fonctions mathématiques (fonctions trigonométriques, logarithmiques...).

```
>>> import math
>>> a = 4.5
>>> math.sin(a)
>>> b = 1e-10
>>> math.log10(b)
```

18.4 Nombres complexes

Python offre la possibilité de manipuler des nombres complexes sans utiliser de module complémentaire.
Un nombre complexe peut être indiqué littéralement en utilisant `j` ou le constructeur **`complex`** :

```
>>> a = 4 + 3j
>>> b = 5 + 0j
>>> a = complex(4, 3)
>>> b = complex(5, 0)
```

Ne pas confondre le `j` complexe et la variable `j`. Ce qui suit ne crée pas de nombre complexe :

```
# Ajoute 5 et le contenu de la variable j
>>> a = 5 + j
# Multiplie le contenu de j par 3 et ajoute 5.
>>> b = 5 + 3 * j
```

Pour utiliser les nombres complexes, il faudrait écrire :

```
>>> a = 5 + 1j
>>> b = 5 + 3j
```

Voici quelques méthodes et accesseurs sur les nombres complexes (le module `cmath` contient d'autres outils) :

1. `sys.float_info` contient des informations sur le codage utilisé.

Type	Ordonné ?	Itérable ?	Modifiable ?
list	Oui	Oui	Oui
tuple	Oui	Oui	Non
set	Non	Oui	Oui
str	Oui	Oui	Non
dict	Non	Oui	Oui
bytearray	Oui	Oui	Oui
frozenset	Non	Oui	Non
bytes	Oui	Oui	Non

TABLE III.2 – Propriétés des collections

```
>>> a = 5 + 4j
>>> a.real
5.0
>>> a.imag
4.0
>>> abs(a)
6.4031242374328485
```

19 Types collections

Les collections : listes, tuples, objets itérables etc. font la force et l'efficacité des langages interprétés modernes. Savoir les manipuler permet de concevoir des programmes concis, élégants et efficaces.

Conteneur ou Collection objet destiné à contenir d'autres objets. Une collection peut être ordonnée (c'est alors une séquence) ou non. Les conteneurs standards sont : **`list tuple set str dict`**...

Séquence collection ordonnée d'éléments indicés par des entiers. Les séquences Python sont par exemple : **`list`**,**`tuple`**,**`str`**. On peut accéder à un élément d'une séquence en précisant entre crochets le numéro d'ordre de l'enregistrement. La numérotation commence à 0. (un objet de type **`range`** se *comporte* aussi comme une séquence)

Modifiable capacité pour un objet (pas forcément une collection) de modifier son contenu sans que l'objet soit recréé. Ce point est délicat, et il est assez spécifique à Python. Dans le cas d'un conteneur, on peut le qualifier de récursivement (ou complètement) non modifiable s'il est non modifiable et que tous les éléments qu'il contient sont récursivement non modifiables. Les termes anglais pour modifiable/non modifiable sont : mutable, immutable.

Type list séquence modifiable d'éléments éventuellement hétérogènes

Type tuple séquence non modifiable d'éléments éventuellement hétérogènes.

Hashable Un objet hashable est un objet récursivement non modifiable. Il peut servir de clé dans un dictionnaire. Les éléments d'un ensemble (**`set`**) doivent être hashables.

Type dict (dictionnaire ou tableau associatif) : collection non ordonnée modifiable d'éléments éventuellement hétérogènes. Un tableau associatif est un type de données permettant de stocker des couples clé/valeur, avec un accès rapide à la valeur à partir de la clé. Celle-ci ne peut bien sûr être présente qu'une seule fois dans le tableau. La clé doit être hashable. Le type **`dict`** fait partie des collections de la catégorie *Mapping* qui associent un objet à un autre.

Type set collection non ordonnée d'éléments hashables distincts.

Itérable capacité, pour une collection, d'égrener ses valeurs, par exemple avec une boucle **`for`** Python. Les objets itérables peuvent être parcourus, mais on ne peut pas nécessairement prendre un élément ou un autre en l'adressant.

La table III.2 donne les propriétés des différents types collection.

19.1 Accéder aux éléments

On peut accéder aux éléments d'une séquence par leurs numéros :

```
>>> l=[1, 3, 5, 7] # création d'une liste
>>> print(l, l[1])
```

```
[1, 3, 5, 7] 3
>>> t=(2, 4, 6, 8) # création d'un tuple
>>> print(t, t[2])
(2, 4, 6, 8) 6
>>> s='Supercalifragilistique' # création d'une chaîne
>>> print(s, s[4])
Supercalifragilistique r
```

Indices négatifs

En Python, il existe un moyen de désigner les éléments d'une séquence en partant de la fin : `s[-1]` est le *dernier* élément de la séquence, `s[-2]` est l'avant-dernier etc... En règle générale, si `k>0`, `s[-k]` vaut `s[len(s)-k]`.

On peut extraire une tranche de séquence, voire une tranche échantillonée[2] :

```
>>> t = list(range(11)) # 0 1 2 ... 10
>>> t[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> t[3:]
[3, 4, 5, 6, 7, 8, 9, 10]
>>> t[3:6]
[3, 4, 5]
>>> t[1:8]
[1, 2, 3, 4, 5, 6, 7]
>>> t[1:8:2]
[1, 3, 5, 7]
>>> t[::2]
[0, 2, 4, 6, 8, 10]
```

Pour les dictionnaires, les numéros sont remplacés par des clés (n'importe quel élément hashable) :

```
>>> cri = {'chien': 'aboie', 'chat': 'miaule', 'caravane': 'passe'}
>>> cri
{'chien': 'aboie', 'chat': 'miaule', 'caravane': 'passe'}
>>> cri['chat']
'miaule'
```

19.2 Agir sur les collections

Lorsqu'on agit sur un objet (`s`), il y a deux façons de le faire : En affectant `s`, comme dans `s=s[:]+s[5]`. Ceci fonctionne que l'objet soit modifiable ou non. On extrait `s[:]` et `s[5]`, et on réalise l'opération `+`, le résultat étant un nouvel objet (il se peut que l'opération `+` ne soit pas possible avec les opérandes données...). Puis `s` référencera (affectation) le nouvel objet, l'ancien n'étant plus référencé par `s` (mais il existe peut être encore, référencé par une autre variable). On peut constater que l'objet référencé par `s` n'est plus le même en contrôlant la valeur `id(s)` avant et après l'opération.

En appelant une méthode qui agit sur l'objet qui est référencé par `s` comme dans `s.sort()`. Dans ce cas, l'identifiant (`id`) de `s` n'est pas modifié et c'est l'objet qui était désigné par `s` qui est modifé (le type de `s` doit donc être modifiable, ce qui implique qu'on peut trier ainsi une liste, mais pas un tuple).

Il existe en outre quelques algorithmes implémentés de deux manières :

- modifiant l'objet d'origine
- retournant un nouvel objet modifié

Par exemple, si `t` est une séquence :

- `t.sort()` trie la séquence `t` sur place (elle doit être modifiable)
- `sorted(t)` ne modifie pas `t` et renvoie une nouvelle séquence triée (`t` n'a pas besoin d'être modifiable)

On trouve de même : `reversed(t)` et `t.reverse()` etc...

2. L'opération d'extraction d'une tranche de tableau se nomme *slicing* en anglais. En Python, elle crée un *nouvel* objet, et non une *vue* sur l'objet d'origine.

19.3 Copier une collection

La copie des types simples ne pose pas de problème particulier. Il faut en revanche être prudent avec les collections. On peut copier une collection ainsi :

```
>>> a = [1, 2, [5, 6, 7]]
>>> b = list(a)
```

Mais c'est une copie superficielle (shallow copy). Si un des éléments est modifiable (si c'est une liste par exemple), ça peut être un problème. Le test suivant permet de comprendre ce qui se passe :

```
>>> a = [1, [1, 2]]
>>> print (id(a), id(a[0]), id(a[1]) )
19360368 9357408 19351600
>>> b = list(a)
>>> print (id(b), id(b[0]), id(b[1]) )
20017520 9357408 19351600
>>> print(a, b)
[1, [1, 2]] [1, [1, 2]]
>>> b[0] = 42
>>> print(a, b)
[1, [1, 2]] [42, [1, 2]]
>>> b[1][0] = 54
>>> print(a, b)
[1, [54, 2]] [42, [54, 2]]
```

Pour obtenir un comportement différent, on dispose d'un module `copy` qui offre une copie en profondeur (deep copy) :

```
>>> import copy
>>> a=[1, 2, [5, 6, 7]]
>>> b=copy.deepcopy(a)
>>> b[2][0] = 3.2
>>> print(a, b)
[1, 2, [5, 6, 7]] [1, 2, [3.2, 6, 7]]
```

19.4 Opérations sur les collections

Les opérations sur les collections sont regroupées dans différentes tables :

- Opérations sur les séquences (listes, tuples, chaînes), table III.3
- Opérations sur les listes, table III.4
- Opérations sur les itérables (listes, tuples, chaînes, dictionnaires, ensembles...), table III.5
- Opérations sur les dictionnaires, table III.6
- Opérations sur les ensembles, table III.7

Méthode	Type retour	Description
`list(sequence)`	list	Renvoie une liste contenant les objets de `sequence`
`x == y`	bool	Retourne `True` si `x` et `y` contiennent les même objets (au sens de `==` pour chaque paire d'objets)
`x >= y`	bool	Vrai si `x` est avant `y` (comparaison élément par élément, comme pour l'ordre lexicographique) ou si `x == y`. Faux sinon.
`x <= y`	bool	Vrai si `x` est après `y` (comparaison élément par élément, comme pour l'ordre lexicographique) ou si `x == y`. Faux sinon.
`x > y`	bool	Vrai si `x` est avant `y` (comparaison élément par élément, comme pour l'ordre lexicographique). Faux sinon
`x < y`	bool	Vrai si `x` est après `y` (comparaison élément par élément, comme pour l'ordre lexicographique). Faux sinon
`x != y`	bool	Vrai si `x` et `y` sont de longueur différente ou si un item de `x` est différent d'un item de `y`. Faux sinon.
`x[i]`	item	Retourne l'item en position `i` de `x`. Si `i` est strictement négatif, vaut `x[len(x)+ i]`
`x[[i]:[j]]`	list	Retourne la tranche d'items de la position `i` à la position `j - 1`. Si `i` n'est pas précisé, il est pris égal à 0. Si `j` n'est pas précisé, il est pris égal à `len(x)`. Les indices négatifs ont la même signification que ci-dessus.
`iter(x)`	iterator	Retourne un itérateur sur `x`
`repr(x)`	str	Représentation *officielle* de `x` sous forme de chaîne de caractères

Table III.3 – Opérations sur les séquences

Méthode	Type retour	Description
`list()`	list	Renvoie une liste vide
`[item[,item]+]`	list	Liste contenant les //items// donnés entre crochets
`del x[i]`	.....	Efface l'objet en position `i` de `x` (et décale les autres éléments)
`x[i] = e`	.....	Affecte l'item `e` à la position `i` de `x`. L'item `i` doit déjà exister (la liste n'est pas étirée).
`x += y`	.....	Modifie la liste `x` en lui ajoutant les éléments de `y`
`x *= i`	.....	Modifie la liste `x` pour qu'elle contienne `i` copies concaténée de la liste originale.
`x.append(e)`	None	Modifie `x` en lui ajoutant `e` comme dernire élément.
`x.count(e)`	int	Retourne le nombre d'apparitions de `e` dans `x`. Les apparitions sont détectées en utilisant `==`.
`x.extend(iter)`	None	Modifie `x` en lui ajoutant les éléments de l'itérable `iter`.
`x.index(e,[i,[j]])`	int	Retourne l'indice du premier élément de `x` égale à `e` (au sens de `==`), situé entre les indices `i` et `j-1`. Lève l'exception `ValueError` si un tel élément n'existe pas.
`x.insert(i, e)`	None	Modifie `x` en insérant `e` de tel sorte qu'il se trouve à l'indice `i`.
`x.pop([index])`	item	Supprime (modifie donc `x`) et renvoie l'élément en position `index`. Si `index` n'est pas précisé, il est pris par défaut égal à `len(x)-1`.
`x.remove(e)`	None	Modifie `x` en supprimant la première occurence de `e`. Lève l'exception `ValueError` si `e` n'est pas trouvé.
`x.reverse()`	None	Modifie `x` en renversant l'ordre de ses éléments.
`x.sort()`	None	Modifie `x` pour qu'il soit trié en utilisant la méthode de comparaison `__cmp__` de ses éléments. La méthode accepte deux paramètres optionels : `key` et `reverse`. Si `reverse` vaut `True`, le tri est fait à l'envers (décroissant). Le paramètre `key` est une fonction qui prend en paramètre un élément et renvoie la valeur qui sera utilisée réellement pour le tri.

Table III.4 – Opérations sur les listes

Méthode	Description
`x + y`	Concaténation
`x * i`	Retourne un *nouvel* itérable contenant `i` copies de `x`
`e in x`	Retourne `True` si `e` est dans `x` et `False` sinon
`all(x)`	Vaut `True` si chaque élément de `x` est évalué à `True`
`any(x)`	Vaut `True` s'il y a au moins un élément de `x` évalué à `True`
`enumerate(x,start)`	Retourne une séquence de tuples de la forme `(index,item)` énumérant les éléments de `x` avec leur position. `start` est un offset appliqué à la position initiale (0).
`len(x)`	Nombre d'éléments de `x`
`max(x, key)`	Retourne le plus grand élémént de `x`. L'argument `key` a le même rôle que pour la fonction `sorted`
`min(x, key)`	Retourne le plus petitélémént de `x`. L'argument `key` a le même rôle que pour la fonction `sorted()`
`reversed(x)`	Retourne un nouvel itérateur contenant les même éléments que `x` mais en sens inverse.
`sorted(x, key, reverse)`	Retourne une liste contenant les mêmes éléments que `x`, mais triés en utilisant la méthode de comparaison `__cmp__` de ses éléments. Si `reverse` vaut `True`, le tri est fait à l'envers (décroissant). Le paramètre `key` est une fonction qui prend en paramètre un élément et renvoie la valeur qui sera utilisée réellement pour le tri.
`sum(x, start)`	Retourne la somme des éléments de `x` augmentée de `start` (qui vaut 0 par défaut)
`zip(x1, ..., xN)`	Retourne un nouvel itérateur contenant des tuples de `N` élément. Chaque élément est pris dans un des itérateurs. Exemple : `zip(('a', 'b', 'c'),(1, 2, 3))` vaudra `(('a', 1), ('b', 2), ('c', 3))`

Table III.5 – Opérations sur les itérables

Méthode	Description
`dict()`	Retourne un dictionnaire
`dict(mapping)`	Construit un dictionnaire à partir d'un objet de type mapping (clé/valeur)
`dict(seq)`	Construit un dictionnaire à partit d'une séquence. Remplace le code `D = {}; for k, v in seq: D[k] = v`
`dict(**kwargs)`	Construit un dictionnaire à partir de paires nom=valeur : `dict(alpha=1,beta="omega")`
`k in D`	Vaut `True` si `k` est une *clé* de `D` et faux sinon
`del D[k]`	Efface la clé `k` du dictionnaire
`D1 == D2`	Retourne `True` si les dictionaires ont les même clés associées aux mêmes valeurs (comparaison avec `==`)
`D[k]`	Renvoie la valeur associée à la clé `k` dans `D`. Si la clé `k` n'existe pas,lève l'exception `KeyError`
`iter(D)`	Renvoie un itérateur sur le dictionnaire
`len(D)`	Renvoie le nombre de clés de D
`D1 != D2`	Vaut `Vrai` si `D1` et `D2` ont des clés différentes ou que certaines clés identiques sont associées à des valeurs différentes
`repr(D)`	Retourne la représentation de D
`D[k] = e`	Associe la valeur `e` à la clé `k`
`D.clear()`	Vide `D` de ses clés
`D.copy()`	Retourne une copie non récursive de `D`
`D.get(k[, e])`	Retourne `D[k]` si `k` est une clé de `D` et `e` sinon (par défaut, `e` vaut `None`
`D.items()`	Retourne une *vue* sur le dictionnaire. Souvent utilisé sous la forme : `for k,v in D.items():...`

Table III.6 – Opérations sur les dictionnaires

Méthode	Description
`set(iter)`	Retourne un ensemble, contenant les éléments de l'itérable
`k in E`	Vaut `True` si `k` est dans l'ensemble et faux sinon
`E.pop()`	Efface un élément au hasard et le renvoie
`E | F`	Ensemble union de deux ensembles
`E ^ F`	Ensemble des éléments qui sont dans `E` ou dans `F`, mais pas dans `E` et `F` (différence symétrique)
`E & F`	Ensemble intersection de `E` et `F`
`E - F`	Ensemble des éléments qui sont dans `E` mais pas dans `F`
`E <= F`	Vaut `True` si `E` est un sous ensemble de `F`

Table III.7 – Opérations sur les ensembles

20 Types modifiables ou non

En Python, un type peut être modifiable ou non. Cette distinction est déroutante lorsqu'on débute.

Type	Modifiable/Non modifiable
int	Non modifiable
float	Non modifiable
tuple	Non modifiable
str	Non modifiable
list	Modifiable
dict	Modifiable
set	Modifiable

Un objet est modifiable si on peut modifier le contenu référencé. Le contenu d'une liste étant une liste de références, et une liste étant modifiable, on peut changer ces références, ce qui autorise à écrire :

```
>>> l=[1, 2, 3]
>>> id(l[0])
9357408
>>> l[0] = 42
>>> id(l[0]) # l'objet référencé n'est plus le même
9358720
>>> l
[42, 2, 3]
```

Un tuple n'étant pas modifiable, les mêmes opérations ne sont pas possible :

```
>>> t=(1, 2, 3)
>>> id(t[0])
9357408
>>> t[0] = 42 # on ne peut pas référencer un autre objet
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

En revanche, si un des éléments d'un tuple est une liste, et qu'on modifie cette liste, alors ça fonctionne... :

```
>>> t=([6, 7, 8], 2, 3)
>>> id(t[0])
30698096
>>> t[0].append(1000)
>>> id(t[0]) # l'objet référencé est toujours le même...
30698096
>>> t
([6, 7, 8, 1000], 2, 3)
```

Une chaîne de caractères n'étant pas modifiable, on ne peut donc pas modifier un de ses caractères :

```
>>> s = 'Pithon'
>>> s[1] = 'y'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

On s'en sort en faisant :

```
>>> s = 'Pithon'
>>> s = s[:1] + 'y' + s[2:]
>>> s
'Python'
```

ou bien :

```
>>> s = 'Pithon'
>>> ls = list(s)
>>> ls
['P', 'i', 't', 'h', 'o', 'n']
>>> ls[1] = 'y'
>>> s=''.join(ls)
>>> s
'Python'
```

21 Chaînes de caractères

Le type correspondant aux chaînes de caractères est **str**. Une chaîne de caractères est une séquence non modifiables, composée de caractères unicodes (voir 2.4).

21.1 Créer et manipuler les chaînes

Une chaîne de caractères peut être délimité par " ou '. Il faut de plus tripler le délimiteur si la chaîne est entrée sur plusieurs lignes :

```
>>> s1 = "Python et les chaînes"
>>> s2 = """Ode à Python
Ô toi, Python !...
"""
```

Le délimiteur doit bien sûr être le même en début et en fin de chaîne.
Deux chaînes de caractères peuvent être concaténées avec l'opérateur +. La longueur d'une chaîne est donnée par la fonction **len** :

```
>>> s1 = 'Python'
>>> s2 = ' et les chaînes'
>>> len(s1)
6
>>> s1 + s2
'Python et les chaînes'
>>> len(s1 + s2)
21
```

Typage dynamique, mais typage fort...

Nous savons que le typage de Python est dynamique. En outre, le typage est fort. Cela signifie que Python ne fera pas de conversion de types pour vous et qu'il faudra être explicite :

```
>>> s1 = 'Python version '
>>> v = 3
>>> s1 + v
Can't convert 'int' object to str implicitly
>>> s1 + str(v)
'Python version 3'
```

On peut employer des caractères spéciaux dans les chaînes. Ces caractères sont généralement échappés par \. En particulier, si la chaîne est délimitée par des guillemets simples, il faudra échapper les guillemets simples qu'elle contient.

```
>>> print('Python ? D\'accord !')
Python ? D'accord !

>>> print("Python ?\nD'accord !")
Python ?
D'accord !
```

Une chaîne étant une séquence, il est possible d'accéder à ses éléments (les caractères) avec l'opérateur `[.]`. En outre, puisque le type chaîne n'est pas mutable, on ne peut pas *modifier* un caractère dans une chaîne, il faut en *recréer* une :

```
>>> s = 'Pithon'
>>> s[1]
'i'
>>> s[1] = 'y'
TypeError: 'str' object does not support item assignment
>>> s[:1] + 'y' + s[2:] # création d'une nouvelle chaîne
'Python'
```

Il est possible d'obtenir le code d'un caractère ou le caractère correspondant à un code particulier avec les fonctions **ord** et **chr** :

```
>>> ord('A')
65
>>> chr(65)
'A'
>>> chr(216) + chr(169)
'Ø©'
```

21.2 Tranformations chaînes ↔ listes

On peut créer une liste à partir d'une chaîne : chaque élément de la liste sera un caractère de la chaîne (et une liste étant modifiable, on pourra modifier un caractère). Les éléments d'une liste de chaînes (que chaque élément soit un seul caractère ou non) peuvent être concaténés en une seule chaîne avec la méthode `join` (à laquelle on précise ce qu'il faut mettre comme caractère entre chaque élément concaténé...) :

```
>>> l = list('Pithon')
>>> l
['P', 'i', 't', 'h', 'o', 'n']
>>> l[1] = 'y'
>>> l
['P', 'y', 't', 'h', 'o', 'n']
>>> "-".join(l)
'P-y-t-h-o-n'
>>> "".join(l)
'Python'
```

21.3 Entrées/sorties standard

Vos programmes peuvent permettre la saisie au clavier avec la fonction **input** :

```
>>> a = input('Entrez votre nom : ')
Entrez votre nom : Bond
>>> print(a, type(a))
Bond <class 'str'>
```

La fonction **input** renvoie une chaîne de caractères, mais on peut la convertir au passage :

```
>>> a = int(input('Entrez votre âge : '))
Entrez votre âge : 3
>>> print(a, type(a))
3 <class 'int'>
```

L'affichage de chaînes formatées peut se faire de plusieurs façons :
La plus simple et la moins souple :

```
>>> a = 43.34456
>>> b = 12
>>> print("Un entier", b, "et un float", a)
Un entier 12 et un float 43.34456
```

Avec la méthode **format** :

```
>>> a = 43.34456
>>> b = 12
>>> print("Un entier {} et un float {}".format(b, a))
Un entier 12 et un float 43.34456
>>> print("Un entier {0} et un float {1}".format(b, a))
Un entier 12 et un float 43.34456
>>> print("Un entier {1} et un float {0}".format(a, b))
Un entier 12 et un float 43.34456
>>> print("Un entier {0:03d} et un float {1:.2f}".format(b, a))
Un entier 012 et un float 43.34
```