

Programmation

Release 3.3.3 (2014)

Thierry MASSART

D/2014/0098/142

8e édition – Tirage 2015-16/2

INFO-F-101_A

En application de l'article 23 du Décret du 31 mars 2004, l'auteur consent à mettre son support de cours obligatoire à la disposition des étudiants régulièrement inscrits. Toute reproduction, communication au public, commercialisation ou autre forme d'exploitation de ce fascicule est interdite et pourra faire l'objet de poursuites disciplinaires, civiles et/ou pénales, conformément à la loi du 30 juin 1994 relative au droit d'auteur et aux droits voisins, à moins qu'il s'agisse d'une reproduction effectuée par un étudiant régulièrement inscrit, dans le cadre de son usage strictement privé d'étudiant, qui ne porte pas préjudice à l'exploitation normale de l'oeuvre.

Toute citation doit mentionner le nom de l'auteur et la source.

Tél. : 02-649 97 80 – Fax : 02-647 79 62 – [Http://www.ulb.ac.be/pub](http://www.ulb.ac.be/pub) – E-mail : sdeconni@ulb.ac.be

« Une main plus ou moins habile ne fait pas seule l'artisan, et la valeur de son travail augmente à mesure que s'élève son niveau intellectuel : la créature humaine n'est pas un simple outil. »

Henri La Fontaine (1854-1943)

Docteur en droit de l'ULB, avocat, sénateur, prix Nobel de la Paix en 1913.

Le label FSC : la garantie d'une gestion responsable des forêts

Les Presses Universitaires de Bruxelles s'engagent !

Les P.U.B. impriment depuis de nombreuses années les syllabus sur du papier recyclé. Les différences de qualité constatées au niveau des papiers recyclés ont cependant poussé les P.U.B. à se tourner vers un papier de meilleure qualité et surtout porteur du label FSC.

Sensibles aux objectifs du FSC et soucieuses d'adopter une démarche responsable, les P.U.B. se sont conformé aux exigences du FSC et ont obtenu en avril 2010 la certification FSC (n° de certificat COC spécifique aux P.U.B. : CU-COC-809718-HA).

Seule l'obtention de ce certificat autorise les P.U.B. à utiliser le label FSC selon des règles strictes. Fortes de leur engagement en faveur de la gestion durable des forêts, les P.U.B. souhaitent dorénavant imprimer tous les syllabus sur du papier certifié FSC. Le label FSC repris sur les syllabus vous en donnera la garantie.

Qu'est-ce que le FSC ?

FSC signifie "Forest Stewardship Council" ou "Conseil de bonne gestion forestière". Il s'agit d'une organisation internationale, non gouvernementale, à but non lucratif qui a pour mission de promouvoir dans le monde une gestion responsable et durable des forêts.

Se basant sur dix principes et critères généraux, le FSC veille à travers la certification des forêts au respect des exigences sociales, écologiques et économiques très poussées sur le plan de la gestion forestière.

Quelles garanties ?

Le système FSC repose également sur la traçabilité du produit depuis la forêt certifiée dont il est issu jusqu'au consommateur final. Cette traçabilité est assurée par le contrôle de chaque maillon de la chaîne de commercialisation/transformation du produit (Chaîne de Contrôle : Chain of Custody – COC). Dans le cas du papier et afin de garantir cette traçabilité, aussi bien le producteur de pâte à papier que le fabricant de papier, le grossiste et l'imprimeur doivent être contrôlés. Ces contrôles sont effectués par des organismes de certification indépendants.

Les 10 principes et critères du FSC

1. L'aménagement forestier doit respecter les lois nationales, les traités internationaux et les principes et critères du FSC.
2. La sécurité foncière et les droits d'usage à long terme sur les terres et les ressources forestières doivent être clairement définis, documentés et légalement établis.
3. Les droits légaux et coutumiers des peuples indigènes à la propriété, à l'usage et à la gestion de leurs territoires et de leurs ressources doivent être reconnus et respectés.
4. La gestion forestière doit maintenir ou améliorer le bien-être social et économique à long terme des travailleurs forestiers et des communautés locales.
5. La gestion forestière doit encourager l'utilisation efficace des multiples produits et services de la forêt pour en garantir la viabilité économique ainsi qu'une large variété de prestations environnementales et sociales.
6. Les fonctions écologiques et la diversité biologique de la forêt doivent être protégées.
7. Un plan d'aménagement doit être écrit et mis en œuvre. Il doit clairement indiquer les objectifs poursuivis et les moyens d'y parvenir.
8. Un suivi doit être effectué afin d'évaluer les impacts de la gestion forestière.
9. Les forêts à haute valeur pour la conservation doivent être maintenues (par ex : les forêts dont la richesse biologique est exceptionnelle ou qui présentent un intérêt culturel ou religieux important). La gestion de ces forêts doit toujours être fondée sur un principe de précaution.
10. Les plantations doivent compléter les forêts naturelles, mais ne peuvent pas les remplacer. Elles doivent réduire la pression exercée sur les forêts naturelles et promouvoir leur restauration et leur conservation. Les principes de 1 à 9 s'appliquent également aux plantations.



Le label FSC apposé sur des produits en papier ou en bois apporte la garantie que ceux-ci proviennent de forêts gérées selon les principes et critères FSC.

® FSC A.C. FSC-SECR-0045

FSC, le label du bois et du papier responsable

Plus d'informations ?

www.fsc.be

A la recherche de produits FSC ?

www.jechedufsc.be



VOTRE AVIS COMPTE !

ÉVALUATION DES ENSEIGNEMENTS

Dès le quadrimestre terminé,
évaluez vos enseignements

Pour :

- Donner une rétroaction à vos enseignants
- Proposer des améliorations
- Participer à l'évolution des enseignements
- Valoriser les activités d'enseignement

Une évaluation
à plusieurs dimensions

Portant sur :

- La prestation des enseignants
- L'organisation de l'enseignement
- Le déroulement des séances
- L'évaluation des apprentissages (examens)

→ www.ULB.be/enseignements/evaluation

L'évaluation institutionnelle des enseignements est organisée par l'ULB.
Elle a lieu dès que les enseignements sont terminés.
Elle se déroule en deux campagnes d'enquête en ligne après les sessions de janvier et de juin.

L'étudiant répond anonymement à un questionnaire pour chaque enseignement auquel il a participé. Chaque questionnaire est analysé et les résultats sont envoyés aux enseignants et à la commission pédagogique facultaire.

Les étudiants et les enseignants ont l'obligation de participer aux évaluations.

Syllabus INFO-F-101 Programmation -

Release 3.3.3 (2014)

Thierry Massart

June 02, 2014

CONTENTS

1	Présentation du cours	1
1.1	Avant-propos	1
1.2	But du cours	1
1.3	Compétences principales visées	2
1.4	Ressources	3
1.5	Pourquoi Python ? (et pas C++, Java, ...)	3
1.6	Un peu de vocabulaire	3
2	Mon premier programme Python	5
2.1	Notions fondamentales	5
2.2	<i>Python</i> pour faire des calculs	5
2.3	Erreur syntaxique	7
2.4	<i>Python</i> pour manipuler du texte	7
2.5	Variables et assignation	8
2.6	print()	9
2.7	Types	10
2.8	input()	11
2.9	Commentaires	11
2.10	Assignation multiple et assignation de tuples	12
2.11	Autres fonctions prédéfinies intéressantes	12
2.12	Mon premier programme complet	13
2.13	Sémantique précise de l'assignation	14
3	Contrôle de flux (instructions if et while)	17
3.1	Conditions et instruction conditionnelle if	17
3.2	Instruction pass	21
3.3	Instruction répétitive while	21
3.4	Instruction for	24
3.5	break et continue	24
4	Définition de nouvelles fonctions	25
4.1	Fonction retournant une valeur	25
4.2	Type des paramètres et du résultat	26
4.3	Fonction ne retournant pas de valeur	27
4.4	Exécution de la fonction appelée (passage de paramètres par valeur)	28
4.5	Espace de noms (namespace), variables locales et globales	30

4.6	Passage de fonction comme argument et fonction lambda	32
4.7	Valeur par défaut pour les paramètres et argument <i>mots-clés</i>	32
5	Chaînes de caractères (strings), tuples, listes et instruction for	33
5.1	Chaînes de caractères (strings)	33
5.2	Tuples	36
5.3	Listes	38
5.4	Instruction for	46
5.5	Autres opérations et méthodes sur les séquences, strings, listes	47
5.6	Compréhension de liste (list comprehension)	48
5.7	Copie de listes complexes et référence cyclique	49
6	Conception d'algorithmes simples	55
6.1	Réurrences simples	55
6.2	Manipulation de séquences	59
6.3	argv et eval	62
6.4	Polygones réguliers avec le module <i>turtle</i>	63
6.5	Création d'un module de gestion de polynômes	64
6.6	Tableaux à plusieurs dimensions	69
7	Ensembles et Dictionnaire	71
7.1	Set	71
7.2	Dictionnaire	72
7.3	Changement de type	78
8	Recherches et tris	83
8.1	Les classiques : recherches	84
8.2	Les classiques : tris	86
9	Notion de complexité et grand O	95
9.1	Motivation	95
9.2	Le grand O	101
9.3	Application des règles de calcul	108
9.4	Complexité des méthodes de manipulation de séquences	113
10	Logique, invariant et vérification d'algorithme	117
10.1	Introduction	117
10.2	Rappel de logique	118
10.3	Exprimer l'état du programme formellement	121
10.4	Preuve partielle et totale	126
10.5	Exemples de preuves d'algorithmes	129
11	Récurivité	133
11.1	Motivation et introduction du concept	133
11.2	Mécanismes	136
11.3	Structures de données récursives	140
11.4	Traduction de fonction récursive en fonction itérative	142
11.5	Gestion de la mémoire à l'exécution	143

12 Fichiers, persistance et Exceptions	157
12.1 Fichiers et persistance	157
12.2 Exceptions	160
13 Classes et objets	167
13.1 Objet et classe	167
13.2 Définition de nouvelles classes	168
13.3 Programmation orientée objet	177
14 Quelques mots sur les langages de programmation	179
14.1 Introduction	179
14.2 Les langages de programmation	185
14.3 Paradigmes des langages de programmation	189
14.4 Histoire des langages de programmation	190
15 Règles de bonnes pratiques pour encoder en Python	199
15.1 PEP	199
15.2 Convention de codage	200
15.3 Règles de bonnes pratiques pour concevoir un programme	202
16 Glossaire	205
17 Aide-mémoire <i>Python 3.2</i> - (version Octobre 2012)	211
17.1 Fonctions	211
17.2 Modules	211
17.3 Opérations et méthodes sur les séquences (str, list, tuples)	212
17.4 Méthodes sur les str	212
17.5 Opérateurs et méthodes sur les listes s	213
17.6 Méthodes sur les dict	213
17.7 Méthodes sur les fichiers	214
17.8 Exceptions	214
Index	215

PRÉSENTATION DU COURS

1.1 Avant-propos

Author Thierry Massart <tmassart@ulb.ac.be>

Version 2.1

Date June 02, 2014

Copyright This document has been placed in the public domain.

1.2 But du cours

Le but premier de ce cours et des exercices associés est de donner aux étudiants une connaissance *active* de l’algorithmique de base et de la programmation structurée. Le formalisme utilisé pour décrire les programmes développés dans ce cours est le langage de programmation *Python*.

Notons que, bien qu’une certaine partie de la *syntaxe* et *sémantique* du langage *Python* soit étudiée, ce cours n’est pas simplement un manuel de programmation *Python*. De nombreuses parties de *Python* n’y sont d’ailleurs pas étudiées.

Les concepts de base *Python* utilisés dans le cadre de ce cours pour écrire un algorithme ou un programme *Python* résolvant un problème algorithmique donné sont peu nombreux. Le cours théorique sert à expliquer ces concepts. Il ne vous est pas seulement demandé de comprendre ces notions de base, mais également de les assimiler et de pouvoir les utiliser pour concevoir et dans une moindre mesure analyser, des programmes *Python* résolvant les problèmes posés. La résolution de nombreux exercices ainsi que les travaux à effectuer vous permettront d’arriver à une connaissance de l’algorithmique beaucoup plus profonde, requise au terme de ce cours. Cette matière doit donc être pratiquée de façon constante à la fois pendant et en dehors des séances d’exercices sur papier et sur machine. Une étude entièrement comprise dans une courte période de temps ne peut pas aboutir au but final requis.

De façon complémentaire, ce cours aborde d’autres aspects essentiels pour maîtriser la matière: nous y verrons comment évaluer l’efficacité d’un algorithme (complexité); nous verrons également comment décrire formellement l’état d’un programme et des bribes de la théorie visant à

prouver qu'un programme est *correct*; nous placerons également *Python* dans le paysage des langages à la fois au niveau historique et pour les concepts et paradigmes manipulés.

Note: Le cours ne vise donc pas à passer en revue d'ensemble des notes mises à votre disposition. L'ordre de présentation ainsi que les éléments présentés au cours peuvent différer de celui donné dans ces pages. L'étudiant est invité à lire ces notes ainsi que les références données ci-dessous, en particulier le [livre de Gérard Swinnen sur Python 3](#) (principalement les 11 premiers chapitres), avant d'assister au cours.

1.3 Compétences principales visées

Si vous avez suivi avec fruit le cours, vous aurez amélioré les compétences suivantes :

- L'apprentissage autonome : vous serez dans une dynamique d'apprentissage autonome et permanent dans ce domaine en constante et rapide évolution qu'est l'informatique ; vous pouvez vous adapter tout au long de votre carrière aux technologies nouvelles. Par exemple, vous apprendrez, en vous référant aux manuels.
- La résolution de problèmes : vous aurez la capacité d'analyser des besoins, de structurer l'information, de concevoir, modéliser et implémenter des solutions pertinentes et efficaces ; de façon plus globale on vous demandera dans les cours d'informatique d'acquérir la "pensée informatique" ("computational thinking") en étant capable de faire des abstractions adéquates pour un problème, et d'allier la théorie à la pratique avec l'ordinateur comme support;
- La communication : vous pourrez comprendre les problèmes posés, et expliquer les solutions proposées ; vous pourrez utiliser la communication scientifique et technique (formalismes mathématiques).

En particulier vous pourrez:

- démontrer une bonne compréhension des concepts de base de *Python* ainsi que lire et comprendre des programmes existants
- analyser un problème simple et proposer une solution informatique pour le résoudre et la mettre en oeuvre en *Python*
- exprimer formellement, en formalisme logique, les fonctionnalités attendues d'un programme informatique
- utiliser des outils informatiques de support à la programmation; exploiter la documentation technique
- réaliser des programmes *Python* corrects et bien structurés
- identifier les cas de test pour valider ces programmes.

1.4 Ressources

- Ces pages
- le [livre de Gérard Swinnen sur Python 3](#) (pour la première partie du cours)
- l'[Université Virtuelle](#) (exercices, anciens examens, codes sources, éventuels compléments de matière)
- [python.org](#) (tutoriel, manuels, download, ...)
- un environnement python 3.2, ainsi qu'un éditeur simple comme [gedit](#)
- le site de test [upylab](#)
- le syllabus d'exercices
- Pour ceux qui veulent faire des calculs numériques : [python2.7](#) avec [pylab](#), [numpy](#), [scipy](#), [matplotlib](#),

1.5 Pourquoi Python ? (et pas C++, Java, ...)

Python est un langage créé début des années nonante par [Guido van Rossum](#)

- de haut niveau multi-paradigmes (programmation impérative, orientée-objet, fonctionnelle,...)
- qui permet d'illustrer rapidement et simplement de nombreux concepts de programmation
- portable
- interprété
- doté d'un typage dynamique fort, d'un "Garbage collector" (ramasse-miettes) et d'une gestion des exceptions

Ces concepts seront expliqués lors du cours.

1.6 Un peu de vocabulaire

Le but de l'informatique est d'effectuer du *traitement de l'information*.

L'*information* est un ensemble d'éléments qui ont une signification dans le contexte étudié. Les *données* d'un problème sont représentées par l'ensemble des informations utilisées pour résoudre ce problème en vue d'obtenir les *résultats* escomptés.

Un *algorithme* n'est pas conçu uniquement pour obtenir un résultat pour une donnée bien précise, mais constitue une *méthode* qui permet, à partir de n'importe quelle autre donnée du même type, d'obtenir le résultat correspondant.

Un problème algorithmique peut être vu comme une fonction f dont l'image de toute donnée d de l'ensemble D des données possibles est un résultat r ($= f(d)$) de l'ensemble R des résultats possibles.

L'algorithme qui résout le problème est la méthode calculant $f(d)$ en un temps fini, pour toute donnée d valide.

Notons qu'il n'existe pas d'algorithme pour tout problème (Il est donc intéressant de *déterminer théoriquement l'existence d'un algorithme pour un problème donné* avant d'essayer de l'écrire effectivement). A côté de cela, il existe souvent de nombreux algorithmes pour résoudre un problème donné; leurs qualités propres permettront de les différencier.

Un algorithme doit être *implémenté* sur un ordinateur. Celui-ci ne possède jamais qu'un nombre limité de *mémoire de stockage* d'information dont la précision est limitée. De ce fait pour résoudre certains problèmes qui en théorie pourraient requérir un calcul trop long, ou une *précision* ou un stockage d'information trop importants, des algorithmes ne donnant qu'une *valeur approchée* du résultat doivent être conçus.

Ecrire un petit programme ou a fortiori développer un gros *logiciel* demande une démarche en plusieurs étapes, appelée *processus de développement* d'un programme, qui peut être divisé en plusieurs phases (partiellement) successives.

1. *Analyse* de ce qui est requis et spécification,
2. *Conception*,
3. *Implémentation*,
4. *Tests et installation*,
5. *Exploitation et maintenance*.

Chaque phase produit des résultats écrits soit sous forme de rapport : spécification de ce qui est requis (*cahier de charges*), *manuel utilisateur*, description du fonctionnement, description succincte ou détaillée de l'algorithme, programme dûment commenté, historique des modifications,

Ce cours n'abordera par tous ces points en détails.

MON PREMIER PROGRAMME PYTHON

See Also:

Lire les chapitres 1, 2 et 6 pour `print()` et `input()` du [livre de Gérard Swinnen sur Python 3](#)

2.1 Notions fondamentales

Nous avons décrit l'algorithme comme étant une méthode de calcul d'une fonction f . Cette description donne une vue extérieure d'un algorithme.

Si nous regardons l'intérieur d'un algorithme, la notion fondamentale est celle d'*action*. Une **action** a une durée finie et un effet bien précis. Chaque action utilise une ou plusieurs entités ; les changements d'**états** de ces entités sont les “marques” laissées par les actions.

Pour pouvoir décrire cela, il faut que chaque action soit descriptible dans un langage:

- Soit une action est simple ; on peut alors la décrire sous forme d'une *instruction*.
- Soit elle est trop complexe pour être décrite en une instruction, elle doit alors être décomposée en plusieurs instructions : le groupement de ces instructions est appelé *processus*. Si chaque instruction d'un processus se suit l'une l'autre dans son déroulement, le processus est dit séquentiel.

Un algorithme ou programme est l'instruction, (au sens large) qui décrit un processus complet. L'ordre d'écriture de l'ensemble des instructions d'un algorithme ou d'un programme est généralement différent de l'ordre d'exécution de celles-ci.

Un programme *Python* décrit un tel processus. Regardons quelques programmes simples pour comprendre son fonctionnement.

2.2 Python pour faire des calculs

Après avoir installé l'environnement *Python* (nous utilisons ici la version 3.2), vous pouvez dialoguer directement au clavier avec l'interpréteur *Python* en tapant depuis la ligne de com-

mande (dans un *shell*) la commande `python3` ou `python` si la version par défaut sur votre ordinateur est la bonne.

Le *prompt* “>>>” invite l'utilisateur à *taper* une instruction *Python*. Toute commande ou instruction est *envoyée* à l'ordinateur en tapant la touche `return`.

La *session* suivante donne un exemple d'utilisation du mode interactif pour faire des calculs simples.

```
litpc30:~ tmassart$ python3
Python 3.2.3 (v3.2.3:3d0686d90f55, Apr 10 2012, 11:25:50)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 5 + 7
12
>>> 2 - 11
-9
>>> 7 + 3 * 2
13
>>> (7+3) * 2
20
>>> 20 / 3
6.666666666666667
>>> 20 // 3
6
>>> 11/3
3.6666666666666665
>>> 9/3
3.0
>>> 3.14 * 5**2
78.5
>>> 3.14159 * 5**2
78.53975
>>> 3.14159 ** 100
5.187410133839704e+49
>>> 9 % 2
1
>>> exit()
litpc30:~ tmassart$
```

Note: les valeurs comme 5, 7, 3.14 sont appelées *constantes* ou *littéraux* (literal).

Note: la fonction `quit()` ou `exit()` permet de clôturer la *session* et revenir au *prompt* (symbole d'invite) du *shell*.

2.3 Erreur syntaxique

Malheureusement l'utilisateur (le programmeur) peut commettre des erreurs (*bugs*) dans ce qu'il demande à l'ordinateur.

Tout d'abord, le texte tapé peut ne pas correspondre à du code *Python* correct:

car le caractère “÷” ne correspond pas à l'opérateur de division en *Python* (il fallait utiliser “/” pour la division ou “//” pour la division *entière* ou *tronquée*).

Comme indiqué par l'environnement *Python*, une erreur de *syntaxe* a été détectée.

Note: Les nombres sont représentés avec la convention anglo-saxonne (avec pour les valeurs très grandes ou très petites, une certaine précision et en notation scientifique (exemple “e+49”)). Les expressions sont évaluées en utilisant les notions d'**associativité** et de **priorité** habituelles ainsi que des parenthèses pour modifier l'ordre d'évaluation. Pour les définitions précises sur le fonctionnement des différents opérateurs, vous devez vous référer aux pages [Python Standard Library](#) du site [python.org](#)

Note: Les opérateurs arithmétiques les plus fréquemment utilisés sont + (addition), - (soustraction), * (multiplication), / (division réelle), // (division entière tronquée), ** (exponentiation), % (modulo) ainsi que += (incrémentement) -= (décrémentement) *= (multiplication par w), /= (division par w).

Note: Attention, les opérateurs habituels sont tous associatifs à gauche sauf l'exponentiation qui est associatif à droite (essayez 2**2**3)

2.4 *Python* pour manipuler du texte

Les valeurs et variables *textes* sont appelées *chaînes de caractères* ou *strings* en anglais (classes `str` *Python*). Une valeur de type *string* est représentée sous forme de texte délimité par des doubles apostrophes "du texte" ou par des simples apostrophes 'un autre texte'.

```
>>> type("bonjour")
<class 'str'>
```

Les opérateurs + et * peuvent être utilisés sur du texte et ont comme sémantique la *concaténation* (mise bout à bout des textes) et la répétition.

```
>>> 'cha'+'peau'
'chapeau'
>>> "ha"*5
'hahahahaha'
```

2.5 Variables et assignation

L'exemple suivant illustre l'exécution d'un programme simple manipulant des *variables*.

```
>>> x = 3
>>> y = 12
>>> z = x
>>> x = 2 * x
>>> y = 2 * y
>>> z = x * y
>>> x, y, z
(6, 24, 144)
```

Le programmeur utilise 3 variables qu'il a nommées *x*, *y* et *z*. Chaque variable est une *référence* à une adresse mémoire où est stocké la valeur correspondante ¹.

Le *nom* (appelé *identificateur*) de chaque variable est choisi assez librement par le programmeur tout en respectant un ensemble de règles précis par rapport à sa syntaxe et en évitant de choisir un mot-clé (voir [livre de Gérard Swinnen sur Python 3](#) pour une explication simple ou [The Python Language Reference](#) pour une définition complète de la syntaxe des identificateurs et la liste complète des *mots-clés Python*).

Par exemple une variable ne peut s'appeler `if` qui, comme on le verra plus tard, est un mot-clé du langage *Python*.

```
>>> if=3
File "<stdin>", line 1
    if=3
    ^
SyntaxError: invalid syntax
```

Un type d'action essentiel en *Python* (et dans la plupart des langages de programmation) est l'*assignation* également appelée *affectation*.

L'opération d'assignation a la forme:

$$v = w$$

où *w* est une expression donnant une valeur et *v* est le nom d'une variable.

Dans une première approximation cette opération consiste à mettre la valeur *w* dans la variable *v*

Ainsi la séquence d'instructions donnera les valeurs suivantes pour les variables, où ? signifie que la variable n'existe pas (et donc que sa valeur est indéfinie):

¹ Plus précisément l'objet qui contient la valeur mais aussi d'autres attributs, comme nous le verrons plus loin.

instruction	x	y	z
x = 3	3	?	?
y = 12	3	12	?
z = x	3	12	3
x = 2 * x	6	12	3
y = 2 * y	6	24	3
z = x * y	6	24	144
(6, 24, 144)	6	24	144

Notons que la dernière instruction `x, y, z` a comme seul effet, dans l'environnement interactif, d'afficher les valeurs correspondantes (ici du tuple `(x, y, z)`).

2.6 print()

L'interpréteur *Python* a un certain nombre de fonctions prédéfinies à sa disposition (voir la liste dans la documentation de la [Python Standard Library](https://docs.python.org/3/) du site python.org). Nous verrons certaines de ces fonctions. Commençons par les fonctions `print()`, `float()` et `input()`.

Quand le programmeur désire, non pas faire des tests ou petits calculs rapides, mais écrire un programme complet, il édite un fichier qui par convention porte un suffixe `.py`, contenant son code *Python*. Je peux par exemple *sauver* le fichier `mon-premier-programme.py` du code de mon premier programme sur disque (ou tout autre moyen de stockage) pour ensuite l'exécuter sur mon ordinateur en tapant la ligne de commande

```
python3 mon-premier-programme.py
```

L'exécution se fait en mode *script* par opposition au mode *interactif*.

Warning: Vos fichiers *Python* doivent être sauves dans l'encodage **UTF-8** (voir livre de Swinnen pour plus d'explications). Nous vous conseillons d'utiliser l'éditeur [gedit](#) pour écrire vos programmes *Python*.

Après un court instant, le *prompt* de la ligne de commande s'affiche ce qui montre que l'exécution est terminée: rien n'a été affiché.

```
litpc30:Exemples-cours tmassart$ python3 mon-premier-programme.py
litpc30:Exemples-cours tmassart$
```

Si l'on veut, en mode script, visualiser un résultat dans la *fenêtre de commande*, le programme doit utiliser l'instruction

```
print()
```

par exemple, si l'on remplace la dernière ligne par:

```
>>> print(x, y, z)
```

on obtient

```
litpc30:Exemples-cours tmassart$ python3 mon-premier-programme-corr.py
6 24 144
litpc30:Exemples-cours tmassart$
```

Note: Le [tutoriel sur python 3.2](#) (section Fancier Output Formatting) illustre les possibilités d'écriture qu'offre le langage

2.7 Types

En *Python*, les objets ont un type. Lors de la création, d'un nouvel objet, son type est défini en fonction de l'objet créé. Plus précisément, *Python* est un langage orienté-objet: on dit que les entités créées sont des objets d'une certaine **classe**.

Les types les plus habituellement manipulés sont entier (integer), réel ou plus précisément nombre à virgule flottante (float), booléen ou logique (boolean), chaîne de caractères (string).

L'instruction *Python*

```
type(x)
```

où *x* est un objet, permet de connaître la classe de *x*.

```
>>> x = 3
>>> y = x / 2
>>> type(x)
<class 'int'>
>>> type(y)
<class 'float'>
```

Les exemples précédents n'utilisent que des objets (de type) entiers et réels. L'exemple suivant manipule d'autres types.

```
>>> t="Ceci est un exemple de texte"
>>> t2="C'est important de faire attention aux apostrophes"
>>> flag = t == t2
>>> print(t, t2, flag)
Ceci est un exemple de texte C'est important de faire attention aux apostrophes False
>>> type(t)
<class 'str'>
>>> type(t2)
<class 'str'>
>>> type(flag)
<class 'bool'>
```

Si la "*traduction*" a un sens, il est possible et peut être utile de prendre une valeur d'un certain type et d'en construire une valeur '*correspondante*' dans un autre type. Par exemple

```
>>> r= 3.14
>>> r2 = int(r)
>>> print(r, r2)
```

```

3.14 3
>>> c = '52'
>>> i = int(c)
>>> type(c)
<class 'str'>
>>> type(i)
<class 'int'>

```

Une première application est la *lecture* de valeurs, comme expliqué dans la section suivante.

2.8 input()

Comme déjà expliqué, un algorithme ou un programme peut être vu comme une méthode pour résoudre un problème spécifié. Généralement, le programme reçoit des *données* et renvoie des *résultats*. L'instruction `input()` permet de lire des données lors de l'exécution du programme.

```
x = input("un texte qui invite l'utilisateur à entrer une
          donnée")
```

Après cette instruction `x` contient le texte envoyé par l'utilisateur du programme et *lu* par le programme.

Prenons l'exemple d'un programme dont le but est de trouver les éventuelles racines d'une équation du second degré. Pour recevoir les trois valeurs réelles on pourra écrire

```

>>> a= float(input('valeur de a : '))
>>> b= float(input('valeur de b : '))
>>> c= float(input('valeur de c : '))

```

`float()` traduit le texte lu en valeur réelle s'il correspond bien à un réel dans une syntaxe correcte

2.9 Commentaires

Un programme *Python* doit être syntaxiquement correct pour être compréhensible par l'ordinateur (l'interpréteur qui le lit et l'exécute); il doit également être lisible pour le programmeur ou toute personne qui aurait envie de comprendre son fonctionnement.

Pour cela, il est important que le programmeur écrive dans son code des *commentaires* pertinents. Une façon de faire est de mettre le caractère `#`: tout ce qui suit sur la ligne ne sera ignoré par l'interpréteur.

```

# calcule le pourcentage de l'heure écoulée
percent = (minute * 100) / 60
v = 5                                     # vitesse en mètres / secondes

```

Pour avoir des commentaires plus long qu'une ligne ou un fin de ligne, les commentaires multilignes peuvent être utilisés

```
"""
    Exemple de commentaire multiligne
    que l'on peut mettre dans un programme Python
    ...
"""
```

Nous verrons que les commentaires multilignes permettent de documenter les fonctions écrites par le programmeur qui pourra redemander ces informations plus tard.

2.10 Assignment multiple et assignment de tuples

Python permet d'assigner plusieurs variables en même temps

```
>>> x = y = 3
```

On peut aussi faire des assignments *parallèles*

```
>>> min, max = 4, 7
>>> x, y = y, x
```

Attention, ici la virgule est un délimiteur des éléments d'une liste et sûrement pas le délimiteur entre la partie entière et la partie fractionnaire d'un nombre (*Python* utilise la convention scientifique et anglaise pour cela).

2.11 Autres fonctions prédéfinies intéressantes

- `abs(x)`
- `dir(objet)`
- `divmod(x, y)`
- `eval(expression)`
- `float(x)`
- `help(x)`
- `input()`
- `int(x)`
- `max(a, b, ...)`
- `min(a, b, ...)`
- `print()`
- `round(x, y)`
- `type(objet)`

Voir définition dans le manuel du [Python Standard Library](http://python.org) du site python.org

Note: Les fonctions `help()` et `dir()` donnent des exemples d'introspection que le programmeur peut utiliser (essayez par exemple `help(print)` après avoir importé le module, en utilisant éventuellement la barre d'espace pour passer à la suite de l'explication et la touche `q` pour terminer).

2.12 Mon premier programme complet

Voici un premier programme complet

```
"""
    Calcule la circonférence et la surface
    d'un cercle de rayon lu sur input
"""
__author__ = "Thierry Massart"
__date__ = "18 août 2011"

from math import pi

rayon= int(input('rayon = '))      # rentrer la donnée
circ = 2.0 * pi * rayon           # calcule la circonférence
surface = pi * rayon**2           # calcule la surface
print('circonférence = ', circ)   # imprime les résultats
print('surface = ', surface)
```

`__author__` et `__date__` sont des variables classiquement utilisées pour identifier l'auteur et la date d'écriture du programme.

2.12.1 Module math

Le module `math` contient de nombreuses constantes et fonctions prédéfinies bien utiles telles que :

`pi cos(x) sin(x) sqrt(x) log(x,b)`

Pour utiliser un module *python*, il faut l'importer; par exemple avec :

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

ou par exemple

```
>>> import math as m
>>> m.cos(m.pi/4)
```

```
0.70710678118654757
>>> m.log(1024, 2)
10.0
```

ou encore

```
>>> from math import pi, cos, log
>>> cos(pi/4)
0.70710678118654757
>>> log(1024, 2)
10.0
```

Warning: *Python* permet de faire l'importation de tous les éléments d'un module (exemple: `from math import *`): ceci est fortement déconseillé car

- ne permet pas de savoir d'où vient les éléments que l'on utilise ensuite,
- peut cacher des éléments existants qui ont le même nom
- rend le code difficile à debugger quand un symbole est indéfini
- peut provoquer des *clashes* entre plusieurs noms

Warning: Apprendre python : Plus important que d'avoir une connaissance encyclopédique de *Python*, il est important de pouvoir trouver l'information que l'on cherche et donc avoir une bonne méthode de recherche d'information. Une recherche du contenu du module `math` via `dir(math)`, `help(math)`, une recherche rapide (quich search) ou approfondie dans la [documentation python](#) (Tutorial, Library Reference, Language Reference) ou un bon livre sur *python* est un bon exemple pour vous entraîner. La méthode expérimentale (tester ce que fait une fonction, ...) constitue un complément important à toutes ces démarches.

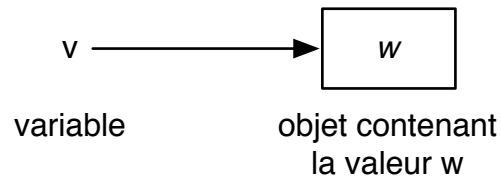
2.13 Sémantique précise de l'assignation

Note: Cette section est cruciale à la compréhension du fonctionnement de *Python*.

L'effet plus précis d'une affectation peut être décomposé en 3 étapes:

- si w n'est pas une référence à un objet déjà existant, évaluer la valeur w et créer un *objet Python* qui mémorise cette valeur ainsi que son type (et d'autres attributs)
- créer et mémoriser un nom de variable v
- établir le lien (par un système interne de pointeur) entre le nom de la variable (v) et l'objet contenant la valeur correspondant à w correspondante. Donc v contient l'adresse en mémoire de l'objet contenant la valeur correspondant à w . Ceci est illustré par une flèche entre v et l'objet.

Le résultat est illustré par le **diagramme d'état** suivant:



Attention: dans une instruction d'assignation, la variable assignée est toujours à gauche sur signe = et la valeur toujours à droite !

Dans l'exemple:

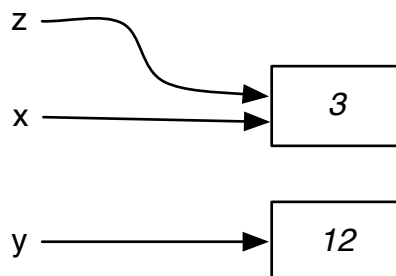
```

>>> x = 3
>>> y = 12
>>> z = x
>>> x = 2 * x
>>> y = 2 * y
>>> z = x * y
  
```

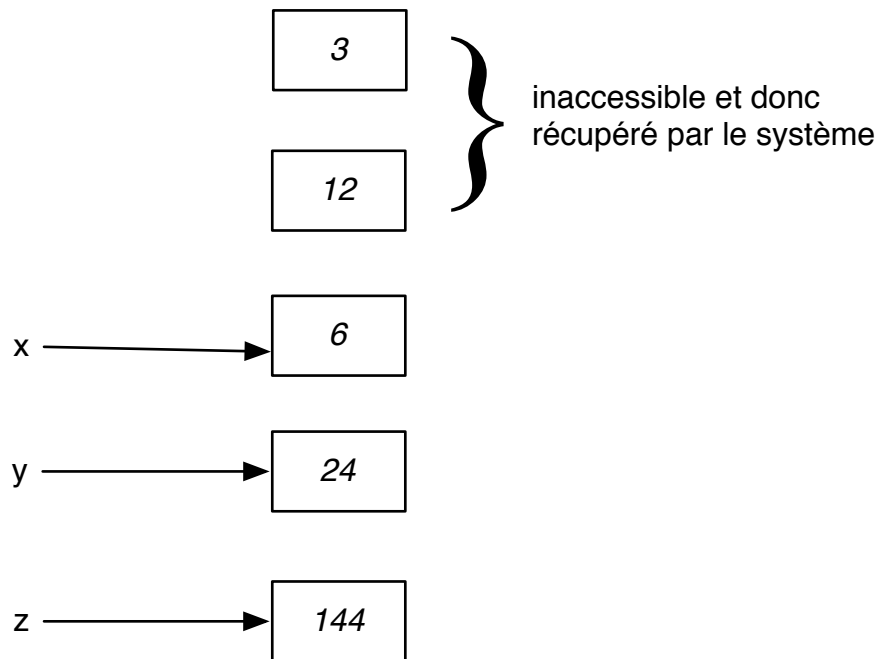
les instructions `x = 3` et `y = 12` créent 2 variables qui référencient respectivement un *objet* de type entier valant 3 et un autre également de type entier valant 12.

L'instruction `z = x` ne crée aucun nouvel objet; mais la variable `z` est créée et reçoit la valeur de `x`, **c'est-à-dire la référence vers l'objet de type entier et valant 3**.

Après ces 3 premières instructions, le *diagramme d'état* suivant illustre la situation:



A la fin du programme entier, 5 objets (de type) entiers ont été créés, dont ceux valant 6, 24 et 144 sont référencés respectivement par les variables `x`, `y` et `z`. Les objets valant 3 et 12 ne sont plus accessibles et donc potentiellement *recupérés* par le système (*garbage collector*).



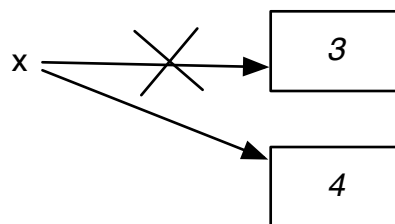
2.13.1 Incrémentation

L'*incrémentation* (respectivement la *décrémentation*) d'une variable x est le fait d'augmenter (resp. diminuer) sa valeur (généralement d'une unité).

le diagramme d'état de

```
>>> x = 3
>>> x = x + 1
```

donne donc:



et implique la création de 2 *objets* de type (classe) `int` : l'un valant 3 et l'autre 4 .

Note: Le fonctionnement précis de l'assignation, sera important pour comprendre le fonctionnement général d'un programme *Python*

CONTRÔLE DE FLUX (INSTRUCTIONS IF ET WHILE)

See Also:

Lire les chapitres 3 et 4 du [livre de Gérard Swinnen sur Python 3](#)

Un programme dans un *langage impératif* comme l'est le langage *Python* exécute une séquence d'instructions dans un certain ordre. Jusqu'à présent, les programmes présentés exécutaient les instruction en séquence.

```
>>> a, b = 3, 7
>>> a = b
>>> b = a
>>> print(a,b)
```

donne un résultat différent de

```
>>> a, b = 3, 7
>>> b = a
>>> a = b
>>> print(a,b)
```

L'ordre d'exécution, appelé *flux d'exécution*, peut être modifié par des instructions dites *composées*. Nous allons voir ici les instructions conditionnelles (*if*) et les instructions répétitives (*while*, *for*). Ce chapitre présente les instructions *if*, *while* et *for*.

3.1 Conditions et instruction conditionnelle *if*

Donnons des exemples simples d'instructions *if* :

```
>>> a = 150
>>> if a > 100:
...     print("a dépasse la centaine")
... 
```

a > 100 est appelée la condition du *test if*. C'est une expression booléenne qui peut donc être évaluée à *vrai* ou *faux*. Si la condition est vraie, ce qui dépend du *if* est exécuté.

```
>>> a = 20
>>> if (a > 100):
...     print("a dépasse la centaine")
... else:
...     print("a ne dépasse pas cent")
... 
```

Reprenons le calcul des éventuelles racines d'une équation du second degré. Après avoir lu les **données** du problème, c'est-à-dire les valeurs a, b et c, il faut calculer le delta et tester s'il est négatif, nul ou positif. Le code du programme donne :

```
"""
    Calcul des racines éventuelles d'une équation du second degré
"""
__author__="Thierry Massart"
__date__="22 août 2012"

from math import sqrt

a= float(input('valeur de a : '))
b= float(input('valeur de b : '))
c= float(input('valeur de c : '))

delta = b**2 - 4*a*c

if delta < 0:
    print(" pas de racines réelles")
elif delta == 0:
    print("une racine : x = ", -b/(2*a))
else:
    racine = sqrt(delta)
    print("deux racines : x1 = ",
          (-b + racine)/(2*a), " x2 = ", (-b - racine) / (2*a))
```

On a donc les trois mots-clés `if`, `else` et `elif`; ce dernier est donc la contraction d'un `else` et d'un `if`. N'oubliez pas de terminer la ligne d'en-tête du `if`, `else`, `elif` par le caractère `:.`

Note: En *Python* il est préférable de mettre une instruction par ligne sauf si l'instruction prend trop de place et prend plusieurs ligne (auquel cas, parfois il est nécessaire de terminer la ligne par le caractère `\` qui indique que l'instruction continue sur la ligne suivante. Le `;` permet également de séparer les instructions d'une même ligne.

L'*indentation* (décalage de certaines instructions de quelques - on conseille 4 - caractères) est essentielle en *Python* car il détermine quelle(s) instruction(s) dépendent de la condition.

Note: Les opérateurs relationnels *Python* sont :

`==, !=, <, >, <=, >=, is, is not, in.`

Note: Les opérateurs booléens *Python* sont :

and, or et not.

La sémantique (valeur) des opérateurs est donnée par leur **table de vérité**, qui exprime le résultat de l'expression en fonction des valeurs des opérandes:

a	b	not a	a and b	a or b
False	False	True	False	False
False	True	True	False	True
True	False	False	False	True
True	True	False	True	True

Les lois de *De Morgan* sont fréquemment utilisées pour simplifier les tests; ayant des expressions logiques A et B

1) `not (A or B) == (not A) and (not B)`

2) `not (A and B) == (not A) or (not B)`

Ainsi à la place de `not ((x >= 0) and (x <10))` on écrira `x < 0 or x >= 10`

3.1.1 If imbriqués

L'indentation est très importante et peut être une source d'erreur en cas d'instructions if (ou while, for) imbriquées.

Le code suivant :

```
if a > 0 :
    if b > 0 :
        print("cas 1")
    else :
        print("cas 2")
```

imprime “cas 1” quand a et b sont positifs, “cas 2” quand a est positif et b négatif ou nul et n’imprime rien si a est négatif ou nul. Le `else` dépend du second `if`.

Le code suivant :

```
if a > 0 :
    if b > 0 :
        print("cas 1")
else :
    print("cas 2")
```

imprime “cas 1” quand a et b sont positifs, “cas 2” quand a est négatif ou nul et donc rien quand a est positif et b négatif ou nul. Le `else` dépend du premier `if`.

3.1.2 Evaluation paresseuse (lazy evaluation)

Lorsqu'une expression booléenne `b1 and b2` est évaluée, si `b1` est fausse, `b2` ne doit pas être évaluée car on sait que le résultat de l'expression est faux.

Idem si `b1 or b2` est évalué avec `b1` évalué à vrai, le résultat est d'office vrai.

Dans ces 2 cas, *Python* n'évalue pas `b2`.

On peut, par exemple, utiliser cela dans le code:

```
>>> if x != 0 and y/x > 1.0:
...     print('ok')
```

3.1.3 Syntaxe générale du if

La syntaxe générale de l'instruction `if` est :

```
"if" expression ":" suite
("elif" expression ":" suite ) *
["else" ":" suite]
```

où

- `expression` est une expression booléenne,
- `suite` un bloc d'instructions (indentées)
- `(...) *` est une méta-notation signifiant que la ligne peut ne pas être mise, mise une ou plusieurs fois
- `[...]` est une méta-notation signifiant que la ligne est optionnelle, c'est-à-dire peut être ou ne pas être mise.

3.1.4 is

l'opérateur `is` peut être utilisé: il teste si deux variables référencent le même objet

```
>>> x=3
>>> y=3
>>> x is y
True
>>> z = x
>>> x is z
True
>>> x = x + 1
>>> x is y
False
```

Ce code montre que l'objet entier contenant la valeur 3 n'est pas créé 2 fois, ce qui serait une perte de temps et d'espace mémoire. Par contre après incrémentation `x = x + 1`, `x` et `y` référencent bien sûr 2 objets différents (`y` vaut 3 et `x` vaut 4).

3.2 Instruction pass

Utilisée généralement quand le code doit être complété, l'instruction `pass` ne fait rien

```
if x < 0:
    pass # TODO : gérer le cas où x est négatif
```

3.3 Instruction répétitive while

Le `while` est l'instruction répétitive la plus simple :

```
i = 3
while i < 10 :
    i = i + 2
    print(i)
print("exemple terminé")
```

imprime :

```
5
7
9
11
exemple terminé
```

3.3.1 Calcul du plus grand commun diviseur de 2 nombres

Un très bel exemple de l'utilisation du `while` consiste en le calcul du plus grand commun diviseur (pgcd) de deux entiers positifs x et y . Par exemple 6 est le pgcd de 102 et de 30 ; nous noterons ceci par

$$\text{pgcd}(102, 30) = 6$$

La méthode du mathématicien grec Euclide est basée sur l'observation que le pgcd de x et y est aussi le pgcd de y et du reste de la division entière de x par y . Ici 12 est le reste de la division entière de 102 par 30 et donc

$$\text{pgcd}(x, y) = \text{pgcd}(y, x \% y)$$

Donc :

$$\text{pgcd}(102, 30) = \text{pgcd}(30, 12)$$

$$\text{pgcd}(30, 12) = \text{pgcd}(12, 6)$$

$$\text{pgcd}(12, 6) = \text{pgcd}(6, 0)$$

Et comme

$$\text{pgcd}(x, 0) = x \text{ (puisque 0 est divisible par tout } x; \text{ pour tout } x : 0.x = 0)$$

nous aurons

$$\text{pgcd}(102, 30) = 6$$

Le programme *Python* calculant le pgcd de 2 nombres entiers positifs est donné à la figure ci-dessous

```
"""
    calcule le pgcd de 2 nombres entiers positifs
"""

x = int(input("Introduisez le premier nombre entier positif : "))
y = int(input("Introduisez le second nombre entier positif : "))

while (y > 0):
    x, y = y, x % y

print("Le pgcd vaut: ", x)
```

La table d'état suivante donne les valeurs des variables durant les différentes phases de l'exécution du programme

instruction	x	y
x = int(input(...	102	?
y = int(input(...	102	30
x,y = y, x%y (1ère it	30	12
x,y = y, x%y (2ème it)	12	6
x,y = y, x%y (3ème it)	6	0
print(...	6	0

3.3.2 Conjecture de Syracuse

```
n = int(input('entier positif : '))
while n != 1:
    print(n, end=' ')
    if n % 2 == 0:
        n = n//2
    else:
        n = n*3+1
```

avec $n = 27795317865557576576432145678$

3.3.3 Autre exemple d'utilisation d'un while

Un autre exemple d'utilisation d'un while est illustré dans le bout de code suivant, qui fait le cumul des valeurs naturelles lues (terminées par une valeur -1 ne faisant pas partie de la liste) :

```
""" Somme cumulée des valeurs positives lues """
res = 0
i = int(input("Donnez une valeur entière (-1 pour terminer) : "))
while i > 0:
    res += i
    i = int(input("Donnez une valeur entière (-1 pour terminer) : "))

print("Somme des valeurs cumulées : ", res)
```

Note: Notons que si la condition est fausse au début de l'exécution d'un `while`, l'instruction ne sera jamais exécutée. Inversément si la condition du `while` ne devient jamais fausse, l'instruction `while` continuera à s'exécuter indéfiniment et ceci jusqu'à ce que le processus qui exécute le programme soit "*tué*" c'est-à-dire arrêté de façon brutale par le programmeur ou le superviseur de l'ordinateur.

Tout programmeur novice va tôt ou tard avoir affaire à un tel programme qui **cycle**. Enfin notons qu'il se peut que la condition devienne fausse au milieu de l'exécution de l'instruction `while`. Ce n'est qu'à la fin de l'exécution de l'instruction que la condition est réévaluée. Ainsi dans le programme *Python* de la figure suivante, le corps de la boucle `while` est exécuté 4 fois.

```
i = 0;
print("Début du programme")
while i < 4 :
    i += 20
    print(i)
    i -= 19
print("terminé")
```

Le format général de l'instruction `while` est :

```
"while" expression ":" suite
["else" ":" suite]
```

où

- `expression` est une expression booléenne,
- `suite` un bloc d'instructions (indentées)
- `[. . .]` est une méta-notation signifiant que la ligne est optionnelle, c'est-à-dire peut être ou ne pas être mise.

Si la partie `else` est présente, elle est exécutée une seule fois quand la condition devient fausse.

3.4 Instruction for

Le `for` est une instruction répétitive qui itère avec une variable de *contrôle* qui à chaque itération, prend une valeur d’une séquence (ou d’un *container* ¹) donnée.

Nous verrons plus d’exemples au chapitre qui présente les strings et les listes. Donnons en seulement une utilisation *classique*:

```
>>> for i in range(10):  
...     print(i)  
...  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

3.5 break et continue

Python a également les instructions `break` et `continue` que nous ne verrons pas ici

Warning: N’utilisez pas d’instruction `break` ou `continue` dans vos programmes. Ces instructions peuvent être évitées et leur utilisation produit des programmes non “*bien structuré*”

¹ Un container est une donnée composée de plusieurs données plus simples (exemple: chaîne de caractère, liste, tuple, dictionnaire)

DÉFINITION DE NOUVELLES FONCTIONS

See Also:

Lire le chapitre 7 du [livre de Gérard Swinnen sur Python 3](#)

Python permet de découper les programmes en fonctions. Chaque fonction créée peut être vue comme un outil résolvant un certain problème. Les autres fonctions peuvent utiliser cet outil sans se soucier de sa structure. Il leur suffit, pour pouvoir l'utiliser, de savoir comment employer cet outil, c'est-à-dire, comme nous le verrons ci-dessous, de connaître le nom de la fonction, les objets (valeurs, variables) qu'elle nécessite et la façon dont elle renvoie un ou plusieurs résultats.

Cette technique a en particulier deux avantages:

- chaque fonction peut (généralement) être testée indépendamment du reste du programme;
- si une partie contient une séquence d'instructions qui doit être réalisée à différents endroits du programme, il est possible de n'écrire cette séquence qu'une seule fois.

En gros, une fonction peut être vue comme une opération dont la valeur est définie par le programmeur. Notons que, dans les chapitres précédents, nous avons déjà vu qu'en *Python*, il existe des fonctions prédéfinies que le programmeur peut utiliser. Ainsi par exemple `print()`, `float()`, `cos()` sont des fonctions; comme on l'a vu, cette dernière est prédéfinie dans le module `math` et doit être importée avant de pouvoir être utilisée.

4.1 Fonction retournant une valeur

Le code suivant définit une fonction `pgcd()` (le programmeur choisit le nom) qui reçoit en *entrée (données)* deux paramètres *formels* (ou juste paramètres) `x` et `y` qui sont supposés ici bien être des valeurs naturelles et renvoie le plus grand commun diviseur de `x` et de `y`.

```
def pgcd(x,y):  
    """Renvoie le plus grand commun diviseur des 2 valeurs reçues  
    qui doivent être des entiers positifs  
    """  
    while y > 0 :
```

```
    x,y = y, x%y
    return x
```

où

```
"""Un commentaire quelconque
   qui peut prendre plusieurs lignes
   """
```

est un commentaire multiligne qui constitue le `docstring` de la fonction `pgcd`.

Une fois définie la fonction `pgcd()` peut être utilisée en l’invokant dans le code. On parle d’*appel* à la fonction et d’*argument* (ou paramètre effectif) passé lors de cet appel. On parle aussi du programme ou la fonction *appelante* et de la *fonction appelée*.

```
a = int(input("valeur entière positive : "))
z = pgcd(26,a)
...
if pgcd(47,z+1) != 1:
    ...
```

On peut aussi demander ce que fait la fonction avec

```
>>> help(pgcd)
```

qui donnera le *docstring* de la fonction.

Rappel: la touche clavier `q` sert à sortir du mode `help`

Note: En informatique, pour spécifier ce que fait un certain code et en particulier une fonction, on parle de *précondition* et de *postcondition*

- Précondition: condition que la fonction suppose vraie pour les entrées, avant d’exécuter son travail
 - Postcondition: condition qu’une fonction garantit si l’ensemble des préconditions sont respectées (=la validité de son résultat)
-

4.2 Type des paramètres et du résultat

Python est un langage interprété avec typage dynamique fort. Cela se remarque par exemple quand on définit une simple fonction `sum`:

```
def sum(a,b):
    return a+b
```

On aura

- `sum(3,5)` renvoie 8 qui est de type `int`
- `sum("bon", "jour")` renvoie *“bonjour”* qui est de type `str` (string)

- `sum("bon", 5)` génère une erreur de type (lève une exception lors de l'exécution de la somme)

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in sum
TypeError: Can't convert 'int' object to str implicitly
```

La raison est que l'on ne peut faire un `+` entre un entier et un string.

Warning: Il est donc impératif en *Python* de bien contrôler les types des objets que l'on manipule et éventuellement de gérer les erreurs possibles si le type reçu ne correspond pas au type ou à un des types attendus. Nous verrons que la gestion des exceptions permet de faciliter la gestion des erreurs.

Note: On dit communément d'une fonction qui renvoie un résultat d'un type `x`, que c'est une fonction de type `x` ou une fonction `x` (par exemple une fonction entière).

Note: Dans le vocabulaire des langages de programmation, une fonction qui peut faire des choses différentes en fonction du type des arguments, est dite *surchargée*. La fonction `sum` est un exemple simple de fonction surchargée qui tantôt effectue une somme, tantôt une *concaténation* de texte.

4.2.1 Fonction booléenne

Il est fréquent de devoir encoder une partie de programme qui réalise un test non élémentaire. La fonction booléenne `est_pair(x)` teste si `x` est pair;

```
def est_pair(x):
    """Renvoie vrai si et seulement si x est pair """
    return x%2 == 0

a = int(input("Entrez un nombre entier impair: "))
if est_pair(a):
    print("il n'est pas impair ! ")
```

4.3 Fonction ne retournant pas de valeur

On peut définir des fonctions dont le but n'est pas de calculer une valeur à renvoyer à la fonction appelante. Voici un simple exemple

```
def print_line(x,n):
    print(x*n)
```

Note: Par défaut une fonction renvoie la valeur `None` qui symbolise l'absence de valeur.

Cela signifie que `return` sans expression donnant une valeur à la fin équivaut à `return None`.

Par ailleurs, un `return` implicite existe à la fin de toute fonction.

Donc après avoir défini `print_line`, le code :

```
>>> print(print_line(5,7))
35
None
```

imprime le résultat de la multiplication et ensuite imprime le résultat de la fonction exécutée (`None`).

De même :

```
>>> print(print_line("*",10))
*****
None
```

imprime 10 fois le caractère “*” et ensuite imprime le résultat de la fonction exécutée (`None`).

4.4 Exécution de la fonction appelée (passage de paramètres par valeur)

Dans la suite, nous parlerons de la fonction appelante pour identifier la fonction qui effectue l'appel à la fonction invoquée; cette dernière sera identifiée comme étant la fonction appelée.

4.4.1 Exécution d'une fonction

Lorsqu'une fonction est appelée par une autre fonction, la séquence du traitement est la suivante:

1. Les variables locales à la fonction appelée, correspondantes aux paramètres formels, sont créées (et mise dans le nouvel espace de nom associé à cette instance de la fonction).
2. Les paramètres sont transmis (passés) entre la fonction appelante et la fonction appelée. Ceci sera décrit en détails ci-dessous.
3. L'exécution de la fonction appelante est suspendue pour laisser la place à l'exécution de la fonction appelée.
4. Lorsque la fonction appelée termine son exécution, les variables locales sont “détruites” : plus précisément l'espace de nom associé à l'instance de la fonction est détruit.

5. La fonction appelante reprend son exécution. Si la fonction précédemment appelée était une fonction retournant une valeur, la valeur de cette fonction a été conservée pour permettre l'évaluation de l'expression utilisant le résultat de cette fonction.

Une fonction ou une procédure étant, en substance, une fonction de traitement paramétrisé, il faut clairement préciser à quoi correspond chaque paramètre formel au début de chacune de ses exécutions.

Tout d'abord, rappelons qu'une fonction est vue comme un opérateur: chaque appel à cette fonction produit un résultat (éventuellement `None`) qui est la valeur de cette fonction pour cet appel ; cette valeur résultat est donc transmise à la fonction appelante. Le passage des paramètres constitue l'autre façon de transmettre des valeurs ou des résultats entre la fonction appelante et la fonction appelée. Le seul mode de transmission des paramètres possible en *Python* est la passage par valeur.

4.4.2 Correspondance des paramètres

Il doit y avoir une correspondance entre les paramètres formels et les paramètres effectifs selon la position. Cela signifie que le *i*ème paramètre effectif en ordre d'apparition dans la liste doit correspondre au *i*ème paramètre formel.

Le passage de paramètre suit le même fonctionnement qu'une assignation: le paramètre effectif est soit une référence à un objet existant soit une expression dont la valeur donne un objet; le paramètre formel est une variable locale qui lors de l'appel recevra une référence à cet objet.

Donc avec `def pgcd(x, y) :` l'appel `pgcd(26,a)` aura comme effet que dans la fonction, `x` référence l'objet entier valant 26 et `y`, la variable référencée par ailleurs par `a`.

4.4.3 Retour de plusieurs valeurs; modification de variables

Un exemple intéressant est celui de la fonction `min_et_max` qui reçoit 2 valeurs et qui trie ces valeurs, par ordre croissant.

La fonction suivante ne donnera pas le bon résultat:

```
def min_et_max(x, y) :  
    """Fonction qui ne fait rien d'utile """  
    if x > y:  
        x, y = y, x  
  
a=int(input("première valeur : "))  
b=int(input("seconde valeur : "))  
  
print(a, b)  
min_et_max(a, b)  
print(a, b)
```

La raison est que `x` et `y` sont des variables locales qui référencent les objets contenant les valeurs, mais que la fonction ne modifie ni les objets, ni les références données lors de l'appel.

La solution est que les valeurs soient renvoyées en résultat.

```
def min_et_max(x,y):  
    """Renvoie min(x,y) suivit de max(x,y) """  
    if x > y:  
        x,y = y,x  
    return x,y  
  
a=int(input("première valeur : "))  
b=int(input("seconde valeur : "))  
  
print(a,b)  
a,b = min_et_max(a,b)  
print(a,b)
```

Warning: Règle de bonne pratique :

Une fonction de type `None` ne doit pas contenir de `return`. Une fonction d'un type différent de `None` ne doit contenir qu'un seul `return` avec le résultat, et ce `return` doit être placée comme dernière instruction de cette fonction.

4.5 Espace de noms (namespace), variables locales et globales

Chaque fois qu'une fonction est appelée, *Python* réserve pour elle (dans la mémoire de l'ordinateur) un nouvel *espace de noms (namespace)* où toutes les variables locales seront mises. Un espace de nom établit une correspondance (mapping) entre des noms (les variables) et des objets référencés.

Rappelons qu'en *Python* une variable `x` contient la référence à un objet `o`; c'est l'assignation qui établit le lien entre `x` et `o`. Un objet, une fois créé, existera tant qu'il sera référencé quelque part.

Rappelons aussi que la *sémantique* de `x = y` où `x` et `y` sont des variables, ne fait que mettre la valeur de `y`, c'est-à-dire la référence vers un objet, dans la variable `x`.

Note: Nous verrons lorsque nous aborderons les types de données *modifiables* à quoi il faudra faire attention.

Chaque module a également son espace de nom. Ainsi `import math` permet de *connaître* le module `math` (en fait de créer un objet `math` de type `class module`) et donc d'accéder à la fonction `cos()` avec la syntaxe `math.cos()` en utilisant la *dot notation*.

Note: Le module par défaut est dénommé `__main__` (avec deux souligné au début et à la fin).

Lors de l'exécution d'une fonction, elle peut accéder aux variables locales ainsi qu'aux variables globales (visibles). Par exemple:

```
>>> def f():
...     p=1
...     print(p,q)
...
>>> p,q=10,20
>>> f()
1 20
>>> print(p,q)
10 20
```

On parle de la *portée (scope)* des variables : zone du programme dans laquelle elle est disponible. La portée d'une variable locale ou d'un paramètre est limitée à sa fonction.

4.5.1 Traceback

Lorsqu'une erreur se produit lors de l'exécution d'un programme *Python* l'utilisateur reçoit un message d'erreur qui commence par `Traceback (most recent call last):` :

Pour identifier où a eu lieu l'erreur, le message *retrace* dans quelle fonction `f0` et où a eu lieu l'erreur ainsi que, si c'est le cas, la fonction `f1` et le lieu où `f0` a été appelée, et ainsi de suite.

Par exemple:

```
def ma_fun():
    une_autre_fun()

def une_autre_fun():
    encore_une_fun()

def encore_une_fun():
    # une_erreur utilisé sans avoir été définie auparavant
    une_erreur= une_erreur + 1

ma_fun()
```

donnera:

```
litpc30:Exemples-cours tmassart$ python3 traceback.py
Traceback (most recent call last):
  File "traceback.py", line 10, in <module>
    ma_fun()
  File "traceback.py", line 2, in ma_fun
    une_autre_fun()
  File "traceback.py", line 5, in une_autre_fun
    encore_une_fun()
  File "traceback.py", line 8, in encore_une_fun
    une_erreur= une_erreur + 1 # une_erreur utilisé sans avoir été définie auparavant
UnboundLocalError: local variable 'une_erreur' referenced before assignment
```

4.6 Passage de fonction comme argument et fonction *lambda*¹

Python permet de passer des fonctions comme paramètre.

```
from math import sqrt, sin
```

```
def fun(f, x):  
    return f(x)
```

```
fun(sqrt, 4)  
fun(sin, 0.5)
```

Cela peut être intéressant ou essentiel dans certain code. Ceci sera illustré dans un chapitre ultérieur dans un code qui dessine une fonction, celle-ci étant donnée en paramètre, dans un certain intervalle.

Par ailleurs, l'aspect dynamique de *Python* permet également de définir des nouvelles fonctions lors de l'exécution et d'avoir des variables qui référencient ces fonctions. En particulier on peut utiliser des définition *lambda*, par exemple

```
>>> f = lambda x : x**2 - 3.0*x  
>>>      # Définit une fonction (x^2-3x) référencée par la variable f  
...  
>>> print(f(2.0)) # appel f(2.0) et imprime le résultat  
-2.0
```

4.7 Valeur par défaut pour les paramètres et argument *mots-clés*

Voir livre de Gérard Swinnen ou manuels.

¹ Matière avancée

CHAÎNES DE CARACTÈRES (STRINGS), TUPLES, LISTES ET INSTRUCTION FOR

See Also:

Lire les chapitres 5 et 10 du [livre de Gérard Swinnen sur Python 3](#)

Python manipule des types de données *non simples* c'est-à-dire où une donnée est formée de parties. Nous allons étudier ici les types de données qui forment des *séquences* : les chaînes de caractères (strings) que nous avons déjà présentées, ainsi que les listes, et les tuples.

5.1 Chaînes de caractères (strings)

Une valeur de type string est entourée des caractères `'...'` ou `"..."` ou `"""..."""`. Les `"""..."""` permettent de taper librement les passages à la ligne, sinon on peut utiliser `\n`

```
>>> a = "bonjour"
>>> b = """ ici je mets
... mon texte
... sur plusieurs lignes"""
>>> print(b)
ici je mets
mon texte
sur plusieurs lignes
```

Nous avons déjà vu que les strings pouvaient être concaténés `a+b` ou répétés `a*5`.

On peut également comparer 2 strings: l'ordre de la codification `unicode UTF-8` est utilisée

Les composantes (caractères) d'un *string* `s` sont indicées de 0 à `len(s)` où `len()` est une fonction prédéfinie qui donne le nombre de caractères dans le string.

Warning: Donc quand `len(s)` vaut 5 on peut manipuler `s[0]` à `s[4]` (`= s[len(s)-1]`).

Python permet aussi d'utiliser des indices négatifs sachant que `s[-1]` est le dernier caractère du string `s`, `s[-2]` l'avant dernier

On peut aussi avoir une *tranche (slice)* de `s` (exemple: `s[2:5]` correspond à la partie de `s` depuis le caractère d'indice 2 compris jusqu'au caractère 5 **non compris**.

`s[:j]` est synonyme de `s[0:j]` et `s[i:]` est synonyme de `s[i:len(s)]`.

Enfin, on peut prendre certains éléments, par exemple :

- `s[::2]` est le string formé des caractères pairs de `s` ou
- `s[::-1]` est le string formé de tous les caractères de `s` mais dans l'ordre inverse.

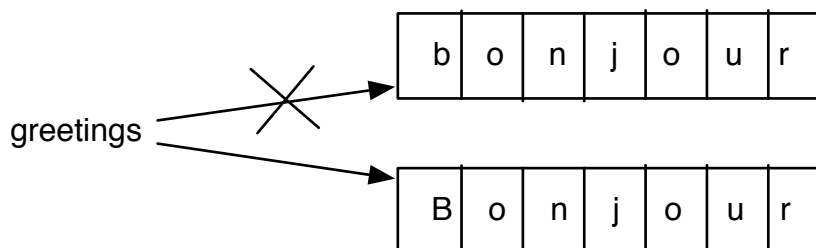
```
>>> s = "bonjour"
>>> print(len(s))
7
>>> print(s[1])
o
>>> print(s[-1])
r
>>> print(s[2:5])
njo
>>> s2 = s[::-1]
>>> print(s2)
ruojnorb
>>> print(s[::2])
bnor
>>> print(len(s))
7
>>> print(s[-len(s)])
b
>>> s3 = 'B'+s[1:]
>>> print(s3)
Bonjour
>>> print(s[:3]*2)
bonbon
>>> vide=""
>>> len(vide)
0
```

Un string est une séquence **immuable**: on ne peut donc pas la modifier

```
>>> s = "bonjour"
>>> s[0]="B"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Ayant "bonjour" dans la variable `greetings`, le code correct pour mettre la première lettre en majuscule est :

```
>>> greetings = "B" + s[1:]
```



5.1.1 L'instruction for ... in

permet de parcourir tous les éléments d'une séquence. Par exemple :

```
nom = "Gilles"
for car in nom:
    print(car + ".", end = '')
print()
```

imprime G.i.l.l.e.s..

Le for ... in peut également être utilisé avec les autres types de séquences (tuples, listes) et également les dictionnaires (voir plus loin dans la matière).

5.1.2 Le test in

permet de tester si un élément appartient à une séquence. Par exemple :

```
car = "a"
voyelles = "aeiouyAEIOUYàâéèêëùîï"
if car in voyelles:
    print(car, " est une voyelle.")
```

imprime a est une voyelle..

De la même façon le in peut être utilisé pour tester l'appartenance dans d'autres *containers* (tuples, listes, dictionnaires).

5.1.3 Comparaison de strings

les opérateurs relationnels <, <=, >, >=, ==, != permettent de comparer les strings en utilisant l'ordre donné par la codification *unicode utf-8*. (voir livre de Gérard Swinnen). Sur la codification *unicode UTF-8*, vous devez juste savoir que les lettres sont codifiées les unes à la suite des autres :

- a, b, c , . . . , z
- A, B, . . . Z
- 0, 1, . . . 9

Notez que les fonctions prédéfinies

- `ord(c)` donne la valeur entière correspondant à la codification du caractère `c`
- `chr(i)` donne le caractère correspondant au code `i`

```
nom = "Joseph"
nom2 = "Josephine"
```

```
if nom < "Caroline":
    print("Caroline c'est ma tortue")
```

```
if nom < nom2:
    print("Josephine passe après !")
```

```
print(ord('0'), ord('9'), ord('A'), ord('B'), ord('a'))
for i in range(20000):
    print(chr(i), end="")
```

imprimera :

```
Josephine passe après !
48 57 65 66 97
```

suivi des 20000 premiers caractères (certains ne sont pas imprimables) dans la codification *UTF-8*.

Notez que certains “caractères” ne sont pas imprimables à proprement parler: par exemple `print(chr(7))` a comme effet d’envoyer le code de signal (beep) au haut-parleur de l’écran. D’autres caractères représentent des pictogrammes: par exemple les codes 9812 à 9823 représentent les pièces du jeu d’échec.

5.2 Tuples

Un tuple est une séquence immuable d’éléments; chaque élément pouvant être du même type ou d’un type différent.

```
mes_nombres_lotto = 7, 21, 27, 32, 35, 41
x=int(input("nombre gagnant : "))
if x in mes_nombres_lotto:
    print("le ", x, " est gagnant et est dans mes nombres ", mes_nombres_lotto)
```

Les tuples peuvent , par exemple, servir pour assigner plusieurs valeurs en une seule assignation multiple

```
a,b = b, a # permute les valeurs de a et de b
```

ou encore quand on veut renvoyer plusieurs valeurs dans une fonction

```
def min_et_max(x,y):
    """ renvoie min(x,y) suivit de max(x,y) """
```



```

    if x > y:
        x, y = y, x
    return x, y

a=int(input("première valeur : "))
b=int(input("seconde valeur : "))

print(a,b)
a,b = min_et_max(a,b)
print(a,b)

```

Notez que habituellement, on entoure les éléments du tuple par des parenthèses pour mieux identifier les éléments

```

...
(x, y) = (y, x)
return (x, y)

```

Les opérateurs + (concaténation) et * (répétition) et in ainsi que le slicing `t[i:j]` fonctionnent comme avec les strings. La comparaison de 2 tuples est similaire à ce qui se passe avec les strings (par exemple `(1, 2, 3) < (1, 2, 3, 4)` est vraie).

L'assignation peut définir un tuple dont les éléments sont les valeurs données à droite du symbole = d'assignation,

```
>>> s = 1, 2, 3
```

C'est ce que l'on appelle le *packing*. L'inverse (unpacking) est également possible, quand ayant une séquence (tuple ou string ou liste,...) de *n* éléments, on demande d'assigner à *n* variables les valeurs de ces *n* éléments:

```

>>> s=(1,2,3)
>>> print("s =", s)
s = (1, 2, 3)
>>> a,b,c = s           # unpack les 3 valeurs dans respectivement a, b et c
>>> print("a = ", a)
a = 1
>>> print("b = ", b)
b = 2
>>> print("c = ", c)
c = 3
>>> d, g = s           # s a 3 composantes !!!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)

```

Notez que l'instruction `d, g = s` demande 2 valeurs mais `s` a 3 composantes, d'où l'erreur.

Les comparaisons, le `in` dans un `for` ou dans une condition ou des fonctions comme `len()` fonctionnent avec tout type de séquences que ce soient des tuples, des strings, ou des listes, ou même des dictionnaires (voir plus loin),

Notons que l'on peut avoir des tuples d'éléments non homogènes et qui peuvent être des élé-

ments non simples comme des strings, listes, tuples, dictionnaires (voir plus loin),

```
>>> mon_tup = 'bon', 'voici', 1, 'exemple', (2, 3)
>>> print(mon_tup)
('bon', 'voici', 1, 'exemple', (2, 3))
>>> len(mon_tup)
5
>>> print(mon_tup[4])
(2, 3)
>>> len(mon_tup[4])
2
>>> a,b,c,d,e=mon_tup
>>> print(a)
bon
```

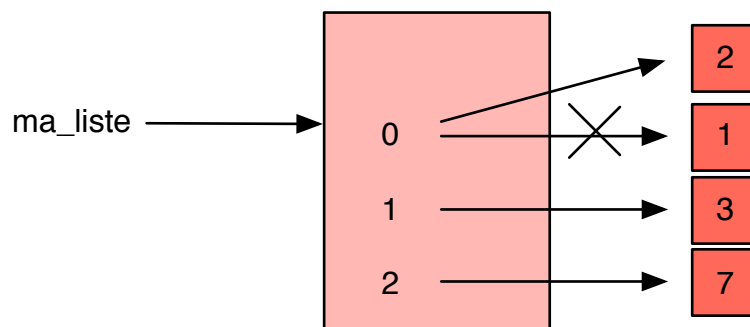
Note: Le tuple contenant un seul élément (par exemple 3 se note (3,)). Le tuple vide de note () (exemple `t = ()`)

5.3 Listes

Une liste *Python* est une séquence que l'on peut modifier (*muable*) d'éléments, chaque élément pouvant avoir le même type ou un type différent des autres :

```
>>> ma_liste = [1, 3, 7]
>>> ma_liste[0] = 2
```

donne le diagramme d'état suivant:



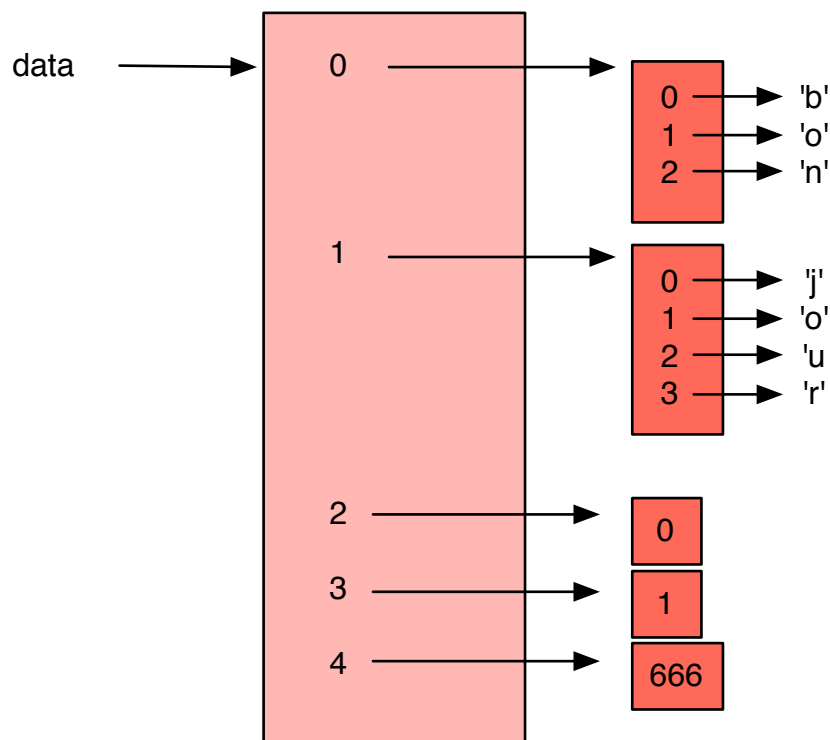
```
>>> ma_liste=[3,7,21,42]
>>> print(ma_liste)
[3, 7, 21, 42]
>>> print(len(ma_liste))
4
>>> empty=[]      # liste vide
>>> type(empty)
<type 'list'>
>>> len(empty)
```

```

0
>>> data=['bon', "jour", 0, 1, [2, 3, 4]]
>>> len(data)
5
>>> type(data[0])
<type 'str'>
>>> type(data[4])
<type 'list'>
>>> data[4]=666
>>> print(data)
['bon', 'jour', 0, 1, 666]
>>> print(data[-1])
666

```

Le diagramme d'état de `data` à la fin de l'exécution du code précédent est :



Ici aussi, les opérateurs `+` (concaténation) et `*` (répétition) et `in` fonctionnent comme avec les strings. La comparaison de 2 tuples est similaire à ce qui se passe avec les strings (par exemple `[1, 2, 3] < [1, 2, 3, 4]` est vraie).

Contrairement aux strings et aux tuples, cette séquence est modifiable: chaque élément (composante) de la séquence peut être modifié. On peut aussi rajouter des éléments ou en supprimer :

```

>>> data=["bon", "jour", 0, 1, [2, 3, 4], (7, 8)]
>>> len(data)
6

```

```
>>> data[0]= "Bonne"
>>> data[1:1]=" "      # insert en composante 1
>>> data[2]="journée"
>>> del data[3] # supprime la composante 3 de data
>>> del data[3:] # supprime les composantes 3 et suivantes de data
>>> print(data)
['Bonne', ' ', 'journée']
>>> len(data)
3
>>> data.append(1515)
>>> print(data)
['Bonne', ' ', 'journée', 1515]
>>> len(data)
4
>>> data.extend([3.14,21,[3,5]])
>>> print(data)
['Bonne', ' ', 'journee', 1515, 3.14, 21, [3, 5]]
>>> len(data)
7
data.insert(1," et heureuse")
>>> print(data)
['Bonne', ' et heureuse', ' ', 'journee', 1515, 3.14, 21, [3, 5]]
>>> print(data[0][0])
'B'
>>> print(data[7][1])
5
```

5.3.1 Copie de liste

```
t = [1,3,7]
s = t      # s et t référencient la même liste
copy = t[:] # copy est une copie de la liste t
t[0] = 36
print(s)
print(t)
print(copy)

imprime

[36, 3, 7]
[36, 3, 7]
[1, 3, 7]
```

Le type de copie que nous venons de présenter s'appelle *shallow* copy (copie superficielle).

Nous verrons plus bas que pour des listes complexes, un shallow copy peut ne pas être suffisant et qu'un *deepcopy* devra être effectué.

5.3.2 Objet et méthodes

Comme déjà brièvement expliqué lorsque nous avons parlé de l'assignation, *Python* manipule des variables qui sont des références à des *objets*. Généralement, associées à une classe d'objets, des procédures de traitement, appelées *méthodes*, manipulent ces *objets* pour les modifier.

Une méthode ressemble à une fonction mais où le traitement est propre à un objet donné.

Le format général pour *exécuter* une méthode `meth` sur un objet `o` est

```
o.meth(...)
```

où à l'intérieur des parenthèses (), sont mis les éventuels arguments

Par exemple:

- `data.append(1515)`
- `data.extend([3.14, 21, [3, 5]])`

Comme le but de ce cours n'est pas de retenir par coeur, l'ensemble des méthodes existantes pour les différentes classes d'éléments, un aide mémoire est toujours mis à votre disposition (Voir le chapitre *Aide-mémoire Python 3.2* et cliquer sur le logo :



en début du chapitre pour obtenir la dernière version *pdf* imprimable sur une feuille A4 recto verso.

5.3.3 Différentes façons de manipuler des listes

Notez que, comme très souvent en *Python* différentes opérations peuvent effectuer le même traitement. Par exemple ayant une liste quelconque `s`,

```
>>> s.append(37)           # rajoute un élément à la fin de la liste t
```

est équivalent à :

```
>>> s[len(s):len(s)] = [37]      # rajoute un élément à la fin de la liste t
```

ou encore à :

```
>>> s.extend([37]) : rajoute à s les éléments de la liste en paramètre
```

ou à :

```
>>> s.insert(len(s), 37) : insère 37 en fin de liste
```

De même:

```
>>> s[0:1] = []           # supprime le premier élément de la liste s
```

est équivalent à :

```
>>> del s[0]          # supprime le premier élément de la liste s
```

Egalement :

```
>>> del s[len(s)-1]   # supprime le dernier élément de la liste s
```

est équivalent à :

```
>>> s[len(s)-1:] = [] # supprime le dernier élément de la liste s
```

Le programmeur a donc intérêt à décider une fois pour toute qu'il utilise toujours la même façon de faire pour réaliser un certain traitement, et à s'y tenir.

Warning: Il est interdit de manipuler en “lecture” (essayer de prendre la valeur de l'élément) ou en “écriture” (essayer de changer la valeur) une composante inexistante d'une liste `s` c'est-à-dire en dehors de l'intervalle `0 .. len(s)`

```
>>> s = ['a', 'b', 'c']
>>> s[3] = 'd'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>> print(s[3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

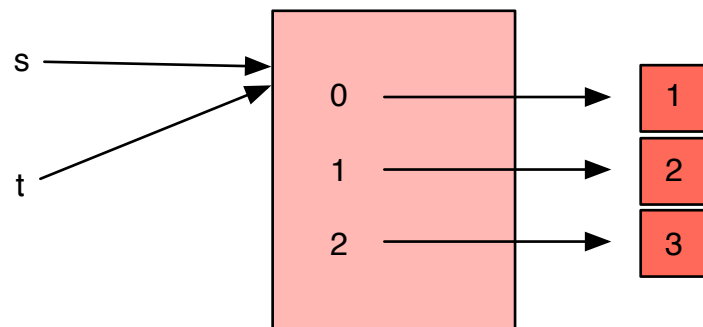
En particulier pour étendre une liste, il faut donc utiliser par exemple une des méthodes vues plus haut (append, extend, slicing).

5.3.4 Liste et références

Warning: Comme pour les valeurs simples l'effet précis des instructions

```
>>> s = [1, 2, 3]
>>> t = s
>>> s is t
True
```

est de créer une liste de 3 objets entiers (valant 1, 2 et 3), de mettre dans `s` la référence à cette liste et ensuite de donner la même référence à `t`; et donc `s is t` répond `True`. Donc `s` et `t` référencent la **même liste** comme le montre le diagramme d'état suivant:



Donc `s[1] = 36` modifie la liste et donc `print(s)` et `print(t)` impriment tous les 2 `[1, 36, 3]`. Quand on utilise `s[i:j] = l2` pour certains indices `i` et `j` et une liste `l2` donnés, ou `del s[i]` ou encore `s.append(v)`, on modifie la liste `s` (et donc `t`). Rappelons que le test `s is t` (qui répond donc `True` ou `False`) permet de savoir si `s` et `t` référencent le même objet.

Si l'on fait par exemple:

```
>>> s = s + [37]
```

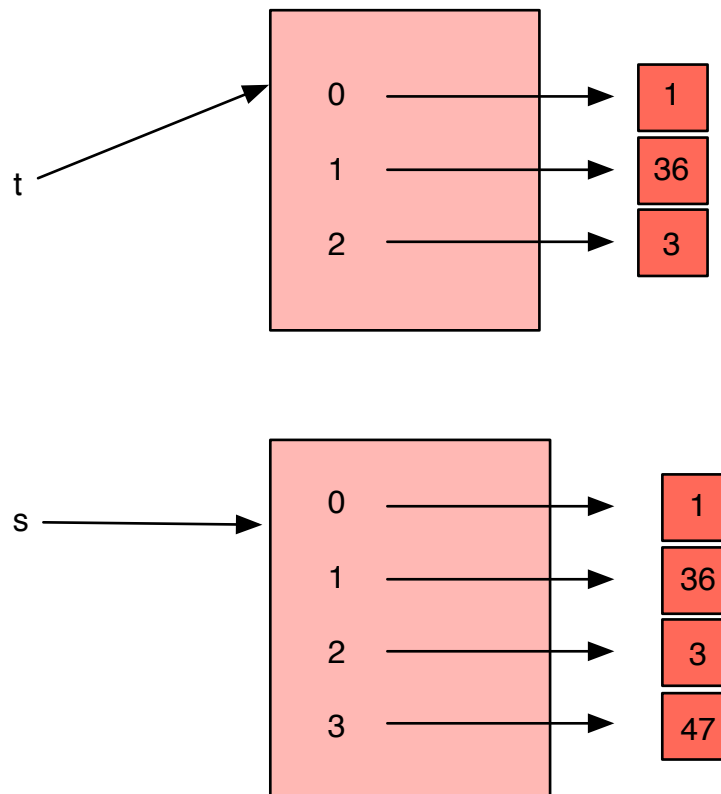
on crée une nouvelle liste avec les éléments de l'ancienne liste `s` et la liste `[37]`. `s` reçoit la référence à la nouvelle liste. Donc `s` référence cette nouvelle liste, `t` référence toujours l'ancienne (`s is t` vaut `False`).

Warning: Cette façon de faire a beaucoup d'avantages, mais est une source d'erreur importante !!

Le code suivant:

```
>>> s= [1,2,3]
>>> t = s
>>> s[1] = 36
>>> s
[1,36,3]
>>> t
[1,36,3]
>>> t is s
True
>>> s = s + [47]
>>> s
[1,36,3,47]
>>> t
[1,36,3]
>>> s is t
False
```

donne en fin d'exécution le diagramme d'état suivant:



Note: Plus précisément, chaque constante (1, 3, 36,...) n'est créée qu'une seule fois (un objet) et si une composante d'une séquence a cette *valeur*, *Python* implémente cela comme une référence vers cet objet ayant cette *valeur* (voir diagrammes d'état suivants).

Warning: Notons que :

```
>>> s = [1,2,3]
>>> t = [1,2,3]
>>> s is t
False
```

crée deux listes distinctes.
tandis que

```
>>> s = 'hello'
>>> t = 'hello'
>>> s is t
True
```

crée un string pointé par les deux variables `s` et `t`.

L'idée est que comme un string est immuable, il n'y a aucun risque à faire pointer plusieurs variables vers le même string, tandis que pour les listes, a priori, elles seront modifiées par la suite et donc il semble clair que dans le code plus haut, `s` et `t` doivent référencer deux listes différentes. Sinon on aurait écrit :

```
s = t = [1,2,3]
```

5.3.5 range()

La fonction `range()` génère une séquence de valeurs entières.

On a déjà vu que

```
for i in range(10):
    print(i, end=" ")
```

imprime : 0 1 2 3 4 5 6 7 8 9

La fonction `range()` est un itérateur, c'est-à-dire une fonction qui génère une séquence d'éléments; `range()` génère une séquence d'entiers.

`range()` peut avoir 2 ou même trois arguments donnant le nombre de départ, la borne et le *pas* qui peut être positif ou négatif. Par défaut le nombre de départ vaut 0 et le pas vaut 1. Les arguments doivent être des nombres entiers.

Notons qu'il est possible d'utiliser la fonction `list()` avec la fonction `range()`, pour obtenir la liste des éléments correspondants.

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

5.4 Instruction for

Généralement, l'instruction `for` sert à assigner à une variable de *contrôle*, une valeur d'une séquence donnée.

Ainsi dans :

```
>>> seq = ['bon', 'jour', 1, 'peu', 2, 'tout']
>>> for elem in seq:
...     print(elem, " / ", end = "")
...
bon / jour / 1 / peu / 2 / tout /
```

la variable `elem` prend à chaque itération une autre valeur de la liste.

Cela fonctionne aussi avec un autre type de séquence telle que un string ou un tuple ou même un dictionnaire comme nous le verrons plus loin.

```
>>> mon_texte = "hello world"
>>> for c in mon_texte:
...     print(c)
...
h
e
l
l
o

w
o
r
l
d
```

Le code suivant illustre comment manipuler l'indice de la séquence, en même temps que l'élément.

```
>>> mon_texte = "hello world"
>>> for i in range(len(mon_texte)):
...     print(i, " ", mon_texte[i])
...
0   h
1   e
2   l
3   l
4   o
5
6   w
7   o
8   r
9   l
10  d
```

Note: Il est possible d'écrire les résultats dans un format plus propre (par exemple, en réservant systématiquement 3 espaces pour une donnée ou en demandant d'imprimer le résultat réel avec une certaine précision). Voir livre de Gérard Swinnen, section “formatage des chaînes de caractères”.

Notez que si on a, par exemple, une liste de strings, on peut avec deux `for` imbriqués, passer en revue tous les caractères de chaque string.

```
>>> ma_liste = ["bon", "jour", " tu vas ", " bien"]
>>> for s in ma_liste:
...     for c in s:
...         print(c, end = " ")
...
b o n   j o u r   t u   v a s       b i e n
```

Le premier `for` assigne à `s` chaque string de `ma_liste`, le `for` imbriqué, assigne à `c` chaque caractère du string mis dans `s`.

5.5 Autres opérations et méthodes sur les séquences, strings, listes

Note: Dans le livre de Gérard Swinnen, et dans l'aide mémoire, vous verrez que d'autres méthodes existent sur les séquences (strings, tuple, listes) ou plus spécifiquement sur un type de séquence (les listes par exemple).

Exemple :

- `del(s[i])`
 - `del(s[i:j])`
 - `s.append(x)`
 - `s.extend(liste2)`
 - `s.count(x)`
 - `s.index(x)`
 - `s.sort()`
 - `s.pop()`
 - `s.reverse()`
 - `s.remove(v)`
-

5.5.1 zip

Cette fonction peut être bien pratique quand on veut parcourir deux ou plusieurs séquences en parallèle. Par exemple le code suivant est équivalent au code vu un peu plus haut :

```
>>> mon_texte = "hello world"
>>> for i, c in zip(range(len(mon_texte)), mon_texte):
...     print(i, " ", c)
...
0   h
1   e
2   l
3   l
4   o
5
6   w
7   o
8   r
9   l
10  d
```

Notez que `zip` peut recevoir plus de deux séquences, et qu'il s'arrête dès que la plus courte est épuisée.

```
>>> mon_texte = "hello world"
>>> for i in zip([1,2,3], "bonjour", "truc"):
...     print(i)
...
(1, 'b', 't')
(2, 'o', 'r')
(3, 'n', 'u')
```

5.6 Compréhension de liste (list comprehension)

La *compréhension de listes* (*list comprehension*) est une méthode efficace et concise pour créer une liste à partir des éléments d'une autre séquence satisfaisant une certaine condition. Le tutoriel python3 dans le site officiel (python.org) donne de très bons exemples d'utilisation de la compréhension de liste.

on peut écrire :

```
squares = [x**2 for x in range(10)]
```

à la place de :

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
... 
```

```
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

De même :

```
>>> combs = [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

est équivalent à :

```
>>>
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Ayant une matrice:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12]]
```

Le code suivant construit sa transposée:

```
>>> trans = [[row[i] for row in matrix] for i in range(4)]
>>> trans
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

5.7 Copie de listes complexes et référence cyclique

On a vu que l'on pouvait faire une copie d'une liste en en prenant une tranche (slicing) depuis le début jusqu'à la fin:

```
s = [1,5,9]
t = s[:] # t référence une nouvelle liste dont les valeurs sont les mêmes
          # que celles de s
```

Malheureusement, stricto sensu, `s[:]` copie les valeurs des composantes de `s` et donc si on a

```
s = [1,5,[7,9]]
t = s[:] # t référence une nouvelle liste dont les valeurs sont les mêmes
          # que celles de s
```

- `t[0]` est une référence vers un objet contenant la valeur entière 1

- `t[1]` est une référence vers un objet contenant la valeur entière 5
- `t[0]` est une référence vers un objet contenant la sous-liste à 2 éléments `[7, 9]`

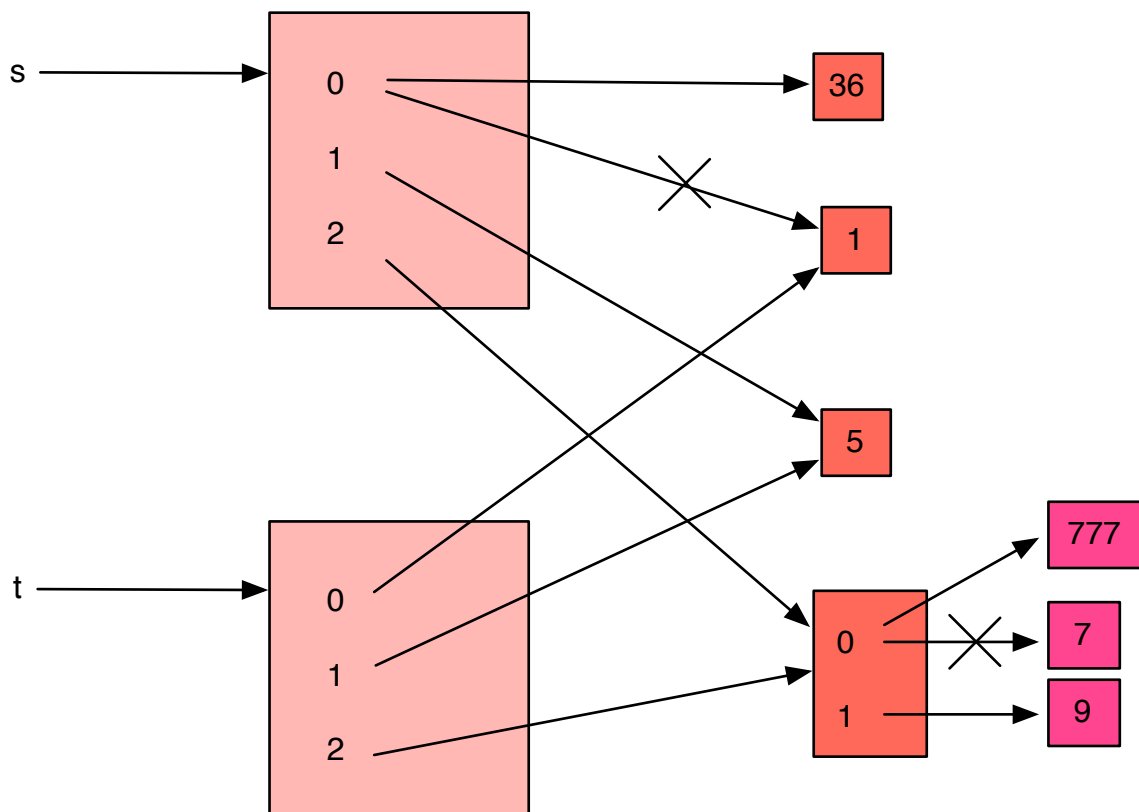
Si le fait que `s[0]` et `s[1]` référencent les mêmes valeurs que respectivement `t[0]` et `t[1]` ne posent pas de problèmes (les valeurs 1 et 5 étant non modifiables), il n'en va pas de même avec la sous-liste `[7, 9]`.

Ainsi on aura :

```
>>> s = [1, 5, [7, 9]]
>>> t = s[:] # t référence une nouvelle liste (shallow copy)
>>> s[0] = 36 # modifie s[0] mais pas t[0]
>>> s[2][0] = 777 # modifie la sous liste pointée par s[2] et t[2] !!
>>> print(s)
[36, 5, [777, 9]]
>>> print(t)
[1, 5, [777, 9]]
```

avec `s[2]` et `t[2]` modifiés de la même façon puisqu'ils référencent la même sous liste.

Le diagramme d'état précis correspondant donne :



La sous liste référencée par `t[2]` (et par `s[2]`) est donc modifiée.

C'est pour cette raison que ce type de copie est dit superficiel (shallow copy).

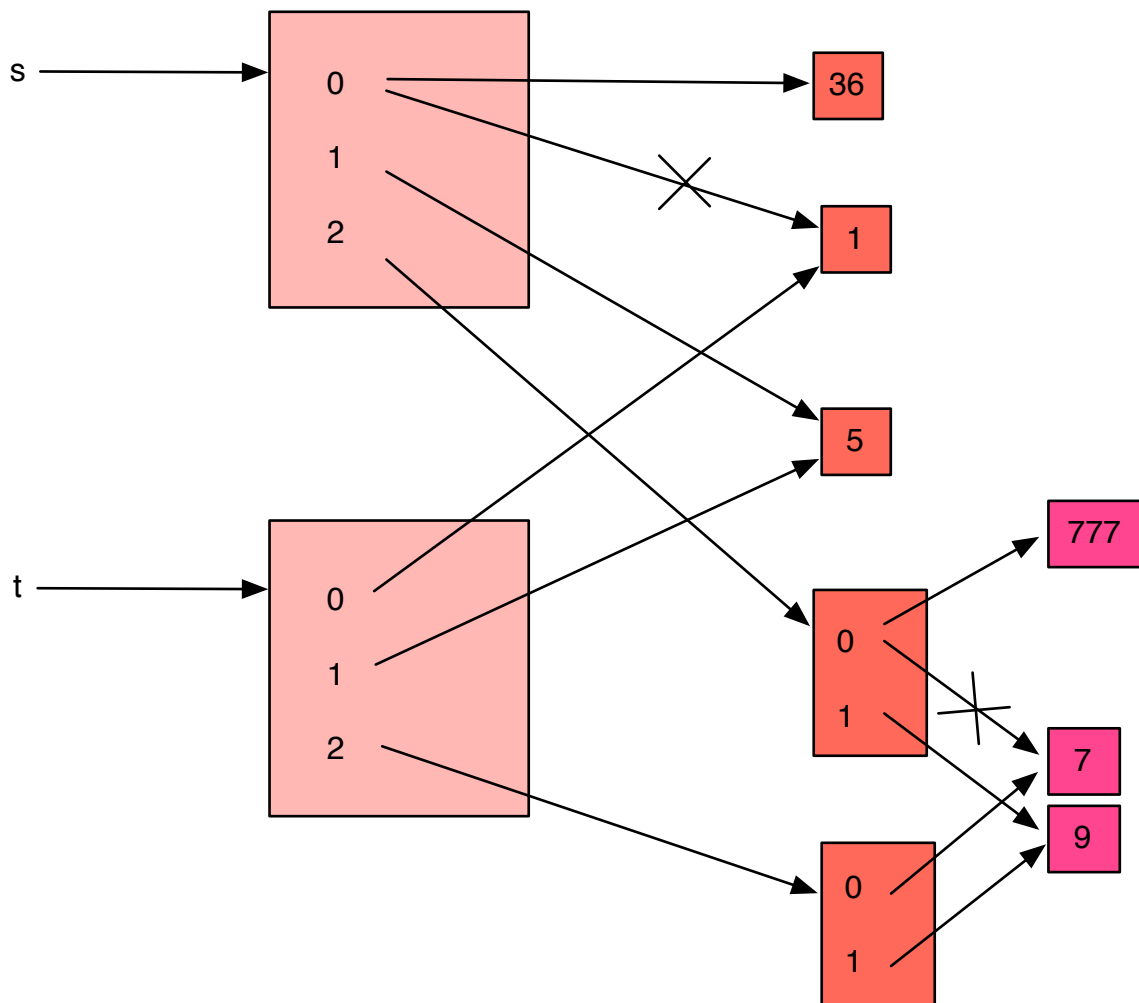
Si l'on désire une structure totalement séparée, même au niveau des sous-listes, la fonction `deepcopy` du module `copy` peut être utilisée :

```

>>> from copy import deepcopy
>>> s = [1, 5, [7, 9]]
>>> t = deepcopy(s) # t référence une nouvelle liste totalement séparée de s
>>> s[0] = 36 # modifie s[0] mais pas t[0]
>>> s[2][0] = 777 # modifie la sous liste pointée par s[2]
>>> print(s)
[36, 5, [777, 9]]
>>> print(t) # t n'a pas été modifié
[1, 5, [7, 9]]

```

Ici la liste `t` n'a pas été modifiée. Le diagramme d'état précis correspondant donne :



Notez que les objets immuables (comme les constantes 1, 5, ...) ne sont pas recopiés puisqu'ici aucun risque n'est possible, étant donné justement qu'ils ne sont pas modifiables.

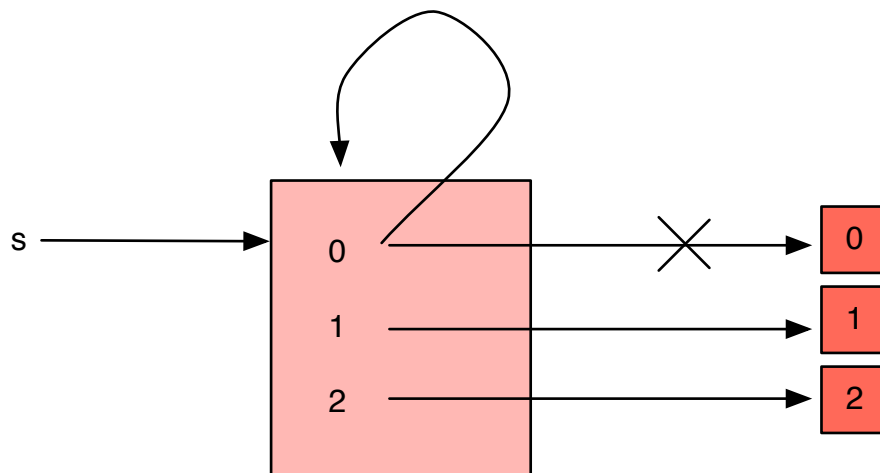
5.7.1 Référence cyclique

le code suivant crée une liste `s` à 3 éléments dont le premier élément référence la liste elle-même.

```
>>> s = list(range(3))
>>> s[0] = s
>>> print(s)
[ [...], 1, 2]
```

Dans le print, le [...] signifie une référence cyclique à une liste sans plus de précision.

Le diagramme d'état correspondant à s en fin de code donne :

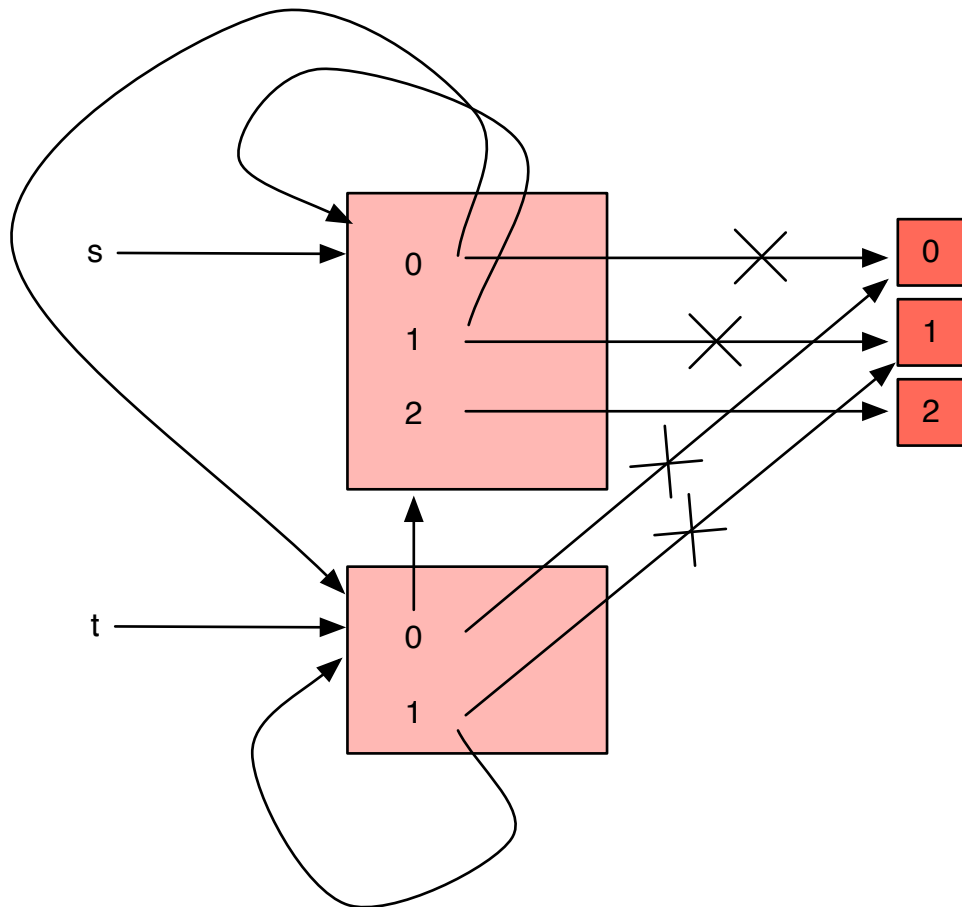


Le code suivant :

```
>>> s = list(range(3))
>>> t = list(range(2))
>>> s[0] = t
>>> t[0] = s
>>> t[1] = t
>>> s[1] = s
>>> print(s)
[[ [...], [...]], [...], 2]
```

Ici, [...] signifie tantôt une référence valant s tantôt une référence valant t.

Ce code donne la structure représentée par le diagramme d'état suivant:



Tout ceci peut vous paraître dans un premier temps assez compliqué: le chapitre suivant revient sur des choses plus abordables en passant en revue des algorithmes utilisant des séquences plus simples.

Note: Le mot **conteneur** (**container**) est utilisé pour signifier un objet qui référence d'autres objets. Les listes, tuples (mais aussi dictionnaires) sont des exemples de conteneurs.

CONCEPTION D'ALGORITHMES SIMPLES

See Also:

Lire les chapitres 5 et 10 du [livre de Gérard Swinnen sur Python 3](#)

6.1 Récurrences simples

6.1.1 Approximation de e

Bien que la bonne connaissance de la syntaxe et de la sémantique d'un langage sont deux éléments indispensables pour écrire un algorithme, la *conception* proprement dite de ce dernier est une tâche souvent bien plus complexe. Pour mener à bien cette tâche, il faut pouvoir analyser et décomposer proprement le problème donné en parties, trouver une méthode de résolution pour chacune de ces parties, traduire ces méthodes dans le langage de programmation, et enfin s'assurer que le programme obtenu est suffisamment *efficace* et répond *correctement* au problème posé: en un mot être **expert en programmation structurée**.

Pour le devenir il faut tout d'abord apprendre à *décomposer* proprement le problème.

Prenons un exemple: supposons que l'on veuille écrire un programme donnant une approximation du nombre e (la constante de Néper). *Python* contient de nombreuses fonctionnalités ou modules qu'il peut importer. Mais ici, faisons l'hypothèse que e n'est pas connu via le module `math`. e peut être défini comme la limite de la série:

$$e = 1 + 1/1! + 1/2! + \dots + 1/n! + \dots$$

Etant donné que le programme ne peut pas s'exécuter pendant une infinité de temps et que la précision de la machine utilisée est finie, il faudra s'arrêter à un terme donné.

Il est connu que l'approximation

$$1 + 1/1! + 1/2! + \dots + 1/n!$$

diffère de e d'au plus deux fois le terme suivant dans la série, c'est-à-dire d'au plus $2/(n+1)!$.

Si l'on estime qu'une précision `eps` donnée (exemple: `eps = 1.0 e-8`) est suffisante, il faut donc écrire un programme qui calcule e avec une erreur inférieure à la constante `eps` c'est-à-dire de sommer les termes $1/i!$ jusqu'à ce que $2/(i+1)! < \text{eps}$.

L'algorithme devra donc calculer :

$$1 + 1/1! + 1/2! + \dots + 1/n!$$

avec $2/n! \leq \text{eps}$ et $2/(n+1)! < \text{eps}$

ce qui ne peut pas être fait en une seule instruction puisque le nombre n n'est pas connu a priori.

Il semble naturel de décomposer l'algorithme en une répétitive qui calcule à chaque tour de boucle un nouveau terme et qui teste si ce nouveau terme doit être inclus dans l'approximation.

Le canevas général sera alors donné par le code suivant :

```
""" Approximation de la valeur de e """

eps = 1.0e-8
n = 1
e = 1.0
terme = 1.0
while 2*terme >= eps:
    e += terme
    n += 1
    ... # calculer le terme 1/n! suivant
print("l'approximation de e = ", e)
```

De plus étant donné que

$$n! = 1.2.3. \dots .n$$

Calculer $n!$ se fait grâce au code:

```
fact_n = 1
for i in range(2, n+1):
    fact_n *= i
```

la valeur $n!$ grandit très rapidement avec n , ce qui peut ne pas être très efficace pour les calculs.

Remarquons que :

```
fact_n = 0
for i in range(1, n+1):
    fact_n *= i
```

$n!$ est évidemment pas correct.

On obtient alors le programme *Python* suivant:

```
""" Approximation de la valeur de e """

eps = 1.0e-8
n = 1
e = 1.0
```

```

terme = 1.0
while 2*terme >= eps :
    # n = prochain indice, e = éval jusque n-1, terme = terme d'indice n
    fact_n = 1
    e += terme
    n += 1
    for i in range(2, n+1):
        fact_n *= i
    terme = 1/fact_n # calcul du terme 1/n! suivant
print("l'approximation de e = ", e)

```

Etant donné qu'au nième tour de boucle on calcule $n!$, on voit aisément que l'on peut diminuer le nombre de calculs: il suffit de reprendre la valeur du terme obtenu au tour précédent et de la diviser par n .

On obtient la version améliorée suivante :

```

""" Approximation de la valeur de e """

eps = 1.0e-8
n = 1
e = 1.0
terme = 1.0
while 2*terme >= eps : # n = prochain indice, e = éval jusque n-1, terme = terme d'indice n
    e += terme
    n += 1
    terme /= n # calcul du terme 1/n! suivant
print("l'approximation de e = ", e)

```

Remarquons, dans le code, que le nom des objets (e , $terme$, n , eps , ...) donne une idée de leur usage. De plus, ce programme est commenté et proprement *indenté*. Ces pratiques facilitent la tâche de toute personne qui doit comprendre le fonctionnement du programme.

Warning: Une chose essentielle également pour comprendre le fonctionnement du `while` est de décrire la “situation” chaque fois que l’on va faire le test (ici: `2*terme >= eps`): ici on explique que lorsque l’on fait ce test, que le résultat soit `True` ou `False`,

- n contient le prochain indice que l’on va considérer pour la série,
- e contient l’évaluation de la série jusque l’indice $n-1$,
- $terme$ contient le terme suivant, c’est-à-dire d’indice n

À chaque itération, cette description est vraie; ce qui change d’une itération à l’autre, est l’indice n ($n += 1$) et donc le fait que l’on avance dans l’évaluation.

6.1.2 Evaluation de la racine carrée d’un nombre

La méthode de Héron permet de calculer la racine carrée d’un nombre a . Si x est une approximation de la racine carrée de a , alors $(x+a/x)/2$ est une meilleure approximation de cette racine.

```
import math

__author__ = "Thierry Massart"
__date__ = "16 september 2011"

EPSILON = 10**-20

def almost_equal(x, y, EPS = 10 ** -7):
    return abs(y - x) < EPS

def square_root(a):
    """Calcul de la valeur approchée de la racine carrée de a
    Par la méthode de Héron
    """
    x = 0.0
    y = 1.0
    while not almost_equal(x, y, EPSILON):
        # rq : pas almost_equal(a, y*y, epsilon) qui peut cycler
        x = y
        y = (y + a / y) / 2
    return y

help(square_root)
print("valeur calculée par la méthode de Héron : \t" \
      "{0:20.18}".format(square_root(2)))
print("valeur calculée par le module math : \t" \
      "{0:20.18}".format(math.sqrt(2)))
```

6.1.3 Somme des nombres pairs dans l'intervalle [i,j]

Ecrire une fonction qui calcule la somme des nombres pairs présents dans l'intervalle [n, m[(n <= m).

```
>>> def sum_even(n, m):
...     if n % 2 == 1:
...         n = n + 1
...     sum = 0
...     for i in range(n, m, 2):
...         sum = sum + i
...     return sum
...
>>> sum_even(2, 5)
6
>>> sum_even(1, 6)
6
```

6.2 Manipulation de séquences

6.2.1 Nettoyage d'un string

La fonction `keepAlnum` retourne la chaîne passée en argument, en lui ayant retiré tous les caractères qui ne sont pas des lettres ou des chiffres.

```
def keepAlnum(s):
    res = ''
    for letter in s:
        if letter.isalnum():
            res = res + letter
    return res

>>> keepAlnum('he23.?56th')
'he2356th'
```

6.2.2 Fonction qui construit la liste des lettres communes à 2 mots

```
def in_both(word1, word2):
    res=[]
    for letter in word1:
        if letter in word2 and letter not in res:
            res.append(letter)
    return res
```

```
print(in_both('pommees', 'oranges'))
```

imprime: ['o', 'e', 's']

6.2.3 Fonction qui lit une liste de valeurs (entières)

La fonction demande à l'utilisateur d'encoder une liste de valeurs entières sur une ou plusieurs lignes, terminée par une ligne vide (return sans rien avant), et renvoie la liste des nombres entiers lus.

La méthode `split` est utilisée pour décortiquer les lignes lues.

```
def lecture(invite):
    res = []
    x = input(invite)
    while x != '':
        decoupe = x.split()
        # liste des parties de x séparées par un/des espace(s) ou tab
        for elem in decoupe:
            res.append(int(elem))
```

```
x = input()
return res

print(lecture("liste d'entiers terminée par une ligne vide : "))
```

6.2.4 Jouer avec les mots

Le fichier words.txt (disponible sur l'Université Virtuelle) contient 113809 mots anglais qui sont considérés comme les mots officiels acceptés dans les mots-croisés. C'est un fichier "plain text", c-à-d que vous pouvez le lire dans n'importe quel éditeur de texte, mais également via Python. Nous allons l'utiliser pour résoudre quelques problèmes sur les mots.

Lire un fichier de texte, ligne par ligne

La fonction open prend en argument le nom d'un fichier, "ouvre" le fichier (par défaut, en mode lecture : mode 'r' (read)), et retourne un objet fichier qui permet de le manipuler.

```
>>> fichier = open('words.txt') # cherche ce fichier dans le repertoire courant
>>> print(fichier)
<open file 'words.txt', mode 'r' at 0x5d660>
```

La méthode readline invoquée sur un objet fichier permet de lire une ligne. Lors de la première invocation, la première ligne est retournée. Lors de l'invocation suivante de cette même méthode, la prochaine ligne sera lue.

```
>>> fichier.readline()
'aa\r\n'
>>> fichier.readline()
'aah\r\n'
```

"aa" est une sorte de lave.

La séquence \r\n représente deux caractères "blancs" (comme le sont espaces, tabulation,...) : un "retour chariot" et une nouvelle ligne, qui sépare ce mot du suivant. L'objet fichier retient l'endroit où l'on se trouve dans la lecture, la seconde invocation permet d'obtenir le mot suivant.

La méthode strip invoquée sur un objet (comme un string) retourne une copie de la chaîne en retirant les caractères "blancs" qui se trouvent au début et en fin de la chaîne.

```
>>> line = fichier.readline()
>>> line.strip()
'aahed'
>>> fichier.readline().strip()
'aahing'
```

Comprenez-vous la dernière instruction ? (Hint : que retourne readline ?)

Puisque la valeur retournée par une méthode est un objet (tout est objet), la notation point permet d'invoquer des méthodes en cascade à lire de gauche à droite : c'est une forme de la composition.

Une boucle for peut être utilisée pour lire les lignes d'un fichier de texte une à une.

```
>>> for line in fichier:
...     print(line.strip())
aahs
aal
aalii
aaliis
...
zymoses
zymosis
zymotic
zymurgies
zymurgy
```

Problème: écrivez un script qui lit le fichier words.txt et qui n'affiche que les mots qui ont plus de 20 caractères (sans compter les caractères blancs).

Solution :

```
fichier = open('words.txt')
for line in fichier:
    word = line.strip()
    if len(word) > 20:
        print(word)
```

Problème: en 1939, Ernest Wright a publié une nouvelle de 50000 mots appelée Gadsby qui ne contient pas la lettre “e”. Comme “e” est la lettre la plus commune en anglais (et dans d'autres langues), ce n'était pas une tâche facile. Ecrivez un script qui n'affiche que les mots qui ne contiennent pas de “e” et calculez le pourcentage de ceux-ci par rapport à l'ensemble des mots du fichier words.txt.

Notez que Georges Perec a fait le même exercice en français, dans son livre La disparition (1969) où il définit ce qui a disparu comme “un rond pas tout à fait clos, fini par un trait horizontal”.

Solution :

```
fichier = open('words.txt')
total = 0
cnt = 0
for line in fichier:
    total = total + 1
    word = line.strip()
    if 'e' not in word :
        print(word)
        cnt = cnt + 1
print('Pourcentage de mots sans e:', cnt / total * 100.0)
```

Problème:

Donnez un mot anglais avec 3 doubles lettres consécutives. Je vous donne un couple de mots qui sont presque candidats, mais pas tout à fait. Par exemple, le mot “committee” serait parfait s'il n'y avait pas ce “i” au milieu. Ou “Mississippi” : si on retirait les “i”, cela marcherait. Il y

a cependant au moins un mot qui possède trois paires consécutives de lettres. C'est peut-être le seul mot, ou il peut y en avoir 500.

Problème:

Ecrire un script qui demande à l'utilisateur le nom d'un fichier de texte (.txt), puis affiche les mots contenus dans le fichier par ordre alphabétique (grâce à une liste). Un même mot ne peut pas être affiché deux fois. Tous les caractères qui ne sont pas des lettres ou des chiffres seront supprimés dans la création de la liste de mots.

Hints :

- la méthode `isalnum` invoquée sur une chaîne retourne vrai ssi tous les caractères de la chaîne sont des lettres ou des chiffres.
- la méthode `split` invoquée sur une chaîne retourne une liste des “mots” de la chaîne (c-à-d les sous-chaînes séparées par des caractères blancs (Espaces, retours à la ligne, tabulations, etc.)). On peut spécifier d'autres séparateurs que des blancs avec le paramètre optionnel `sep`.

Solution :

```
filename = input('Nom du fichier: ')
file = open(filename)
wordsList = []
for line in file:
    for word in line.split():
        cleanWord = keepAlnum(word)
        if not cleanWord in wordsList:
            wordsList.append(cleanWord)
wordsList.sort()
print(wordsList)
```

6.3 argv et eval

6.3.1 argv

La fonction `input` permet d'obtenir des entrées de l'utilisateur. Un autre moyen (rapide et donc souvent apprécié à l'utilisation) d'obtenir des entrées de l'utilisateur, est de récupérer les “arguments de la ligne de commande”.

Exemples :

De la même manière que la commande **shell** `cd myDir` vous permet de changer de répertoire dans une console (le nom du répertoire *myDir* est un argument de la commande `cd`), la commande `python3 myscript.py` vous permet d'interpréter le script *Python3* `myscript.py`.

Tous les arguments passés après la commande *python3* (et séparés par des espaces ¹) sont disponibles via l'attribut `argv` du module `sys`. La valeur `argv` est une liste qui contient ces arguments.

¹ Pour passer un argument qui contient des espaces, on peut le mettre entre apostrophes.

Ainsi pour le script `args.py` suivant :

```
import sys
print(sys.argv)
```

La commande

```
python3 args.py un deux trois "x + y"
```

initialise la liste `sys.argv` à :

```
['args.py', 'un', 'deux', 'trois', 'x + y']
```

6.3.2 eval

Python étant un langage interprété, fournit une fonction `eval(exp)` qui évalue le string `exp` donné en paramètre comme s'il s'agissait de code supplémentaire.

Cela permet par exemple d'écrire un script *Python* qui reçoit en paramètre un polynôme `p` et une valeur `x` et calcule la valeur `p(x)`.

```
"""
    utilisation : python3 eval-poly.py p x
    où p représente un polynôme en format python (ex: 3*x**2+2*x-2)
    sans espace entre les valeurs et les opérateurs
    x donne la valeur pour laquelle on veut la valeur p(x)
"""
import sys

p = sys.argv[1]
x = float(sys.argv[2])

print(eval(p))
```

6.4 Polygones réguliers avec le module *turtle*

Turtle est un module permettant de faire des dessins simples (lignes). Les principales fonctions mises à votre disposition dans le module *turtle* sont les suivantes :

Commande	Effet
reset()	On efface tout et on recommence
goto(x, y)	Aller à l'endroit de coordonnées x, y
forward(distance)	Avancer d'une distance donnée
backward(distance)	Reculer
up()	Relever le crayon (pour pouvoir avancer sans dessiner)
down()	Abaissier le crayon (pour recommencer à dessiner)
color(couleur)	couleur peut être 'red' , 'blue', etc.
left(angle)	Tourner à gauche d'un angle donné (exprimé en degrés)
right(angle)	Tourner à droite
width(épaisseur)	Choisir l'épaisseur du tracé
begin_fill()	Début de la zone fermée à colorier
end_fill()	Fin de la zone fermée à colorier
write(texte)	texte doit être une chaîne de caractères

Après avoir installé *turtle* et fait `import turtle`, n'hésitez pas à utiliser `help(turtle.write)` par exemple pour avoir les détails des fonctions (tailles des caractères écrits, alignement, ...).

Ecrire une fonction `poly(n,long,t)` qui reçoit deux entiers *n* et *long* et le module *turtle* et qui dessine un polygone régulier à *n* côtés, chacun de longueur *long*.

```
import turtle

def poly(n, long, t):
    for i in range(n):
        t.forward(long)
        t.left(360/n)

turtle.reset()

for i in range(3,12):
    poly(i, 50, turtle)

x=input('tapez return pour terminer : ')
```

Modifiez votre code pour produire des *étoiles* à un nombre impair (et ensuite pair) de branches.

6.5 Création d'un module de gestion de polynômes

6.5.1 Encodage d'un polynôme

Une façon simple d'encoder un polynôme :

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

est d'avoir une liste *p* contenant les coefficients, c'est-à-dire avec `p[i]` qui contient le coefficient *a_i* (pour *i* dans 0.. *n*)

Il devient alors assez facile d'effectuer les opérations habituelles sur des polynômes.

6.5.2 Opérations sur les polynômes

Voici les codes pour les opérations habituelles sur un ou des polynômes :

```
def peval(plist,x):
    """
    Eval le polynôme plist en x.
    """
    val = 0
    for i in range(len(plist)): val += plist[i]*x**i
    return val

def add(p1,p2):
    """ Renvoie la somme de 2 polynômes."""
    if len(p1) < len(p2):
        p1,p2 = p2,p1
    new = p1[:]
    for i in range(len(p2)): new[i] += p2[i]
    return new

def sub(p1,p2):
    """ Renvoie la différence de 2 polynômes
    """
    return add(p1,mult_const(p2,-1))

def mult_const(p,c):
    """ Renvoie p multiplié par une constante"""
    res = p[:]
    for i in range(len(p)):
        res[i] = res[i]*c
    return res

def multiply(p1,p2):
    """ Renvoie le produit de 2 polynômes"""
    if len(p1) > len(p2):
        p1,p2 = p2,p1
    new = []
    for i in range(len(p1)):
        new = add(new,mult_one(p2,p1[i],i))
    return new

def mult_one(p,c,i):
    """Renvoie le produit du polynôme p avec le terme c*x^i
    """
    new = [0]*i #termes 0 jusque i-1 valent 0
    for pi in p:
        new.append(pi*c)
    return new

def power(p,e):
    """Renvoie la e-ième puissance du polynôme p"""
```

```
    assert type(e) is int " s'assure que e est entier
    new = [1]
    for i in range(int(e)):
        new = multiply(new,p)
    return new

def derivee(plist):
    """Calcule la dérivée du polynôme dans plist.
    """
    new = []
    for i in range(1,len(plist)):
        new.append(i*plist[i])
    return new

def integrale(plist, c=0):
    """Renvoie le polynôme integrale de plist avec la constante c
    (0 par défaut).
    """
    new = [c]
    for i in range(len(plist)):
        v = plist[i]/(i+1.0)
        new.append(v)
    return new
```

6.5.3 Plot de polynômes avec *turtle*

Le module *turtle*, associé à la fonction `eval()` et à l'importation des paramètres via `argv` avec le module `sys`, permet facilement de dessiner (plot) la valeur d'un polynôme dans un certain intervalle.

Le code suivant utilise *turtle* pour tracer une fonction donnée en paramètre à la commande, en traçant des sections de droite entre les points connexes pour toutes les valeurs x , $f(x)$ avec x valeurs entières dans l'intervalle $[-100, 100]$.

la commande sera par exemple

```
python simple_dessin_f.py "100*cos(x/10)+x"
```

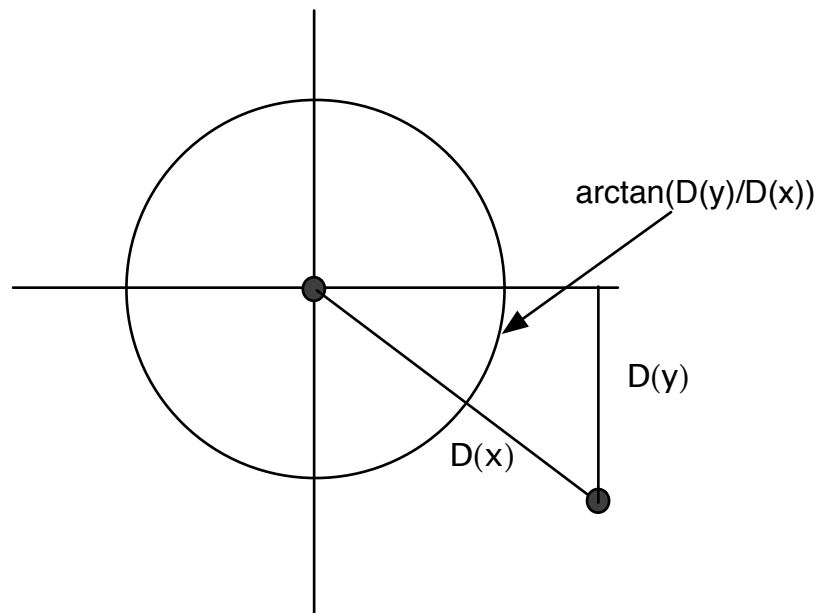
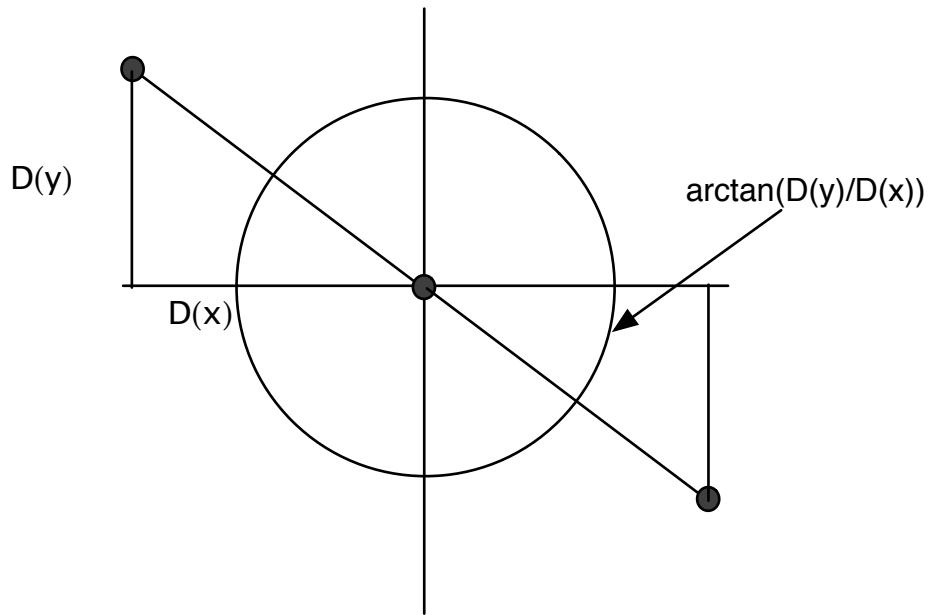
Notez que nous avons exceptionnellement fait `from math import *` dans le code pour permettre à l'utilisateur d'utiliser librement les fonctions mathématiques disponibles dans le module `math`.

Notez que *turtle* affiche une fenêtre dans les valeurs environ $(-300, -300)$ à $(300, 300)$.

La difficulté principale du code dans `simple_dessin_f.py` est de calculer l'orientation et la longueur des déplacements que doit effectuer la *tortue*.

- **La distance euclidienne entre les anciennes valeurs des coordonnées** (old_x, old_y) et les nouvelles (x, y) est utilisée.
- `arc tangente $atan(diff_y/diff(x))$` est utilisée pour calculer l'angle (r : pour mettre la tortue en position initiale, comme on doit déplacer la tortue vers la gauche, et donc

on rajoute π à l'angle). Voir dessin.



```
import sys
from math import *
import turtle

def dessin_fun(t, f):
    """
        exemple simple de tracé d'une fonction entre 2 bornes (ici -100,100)
    """
    t.radians()
    borne_inf = -100 # valeur imposée pour simplifier le code
    borne_sup = 100 # valeur imposée pour simplifier le code
```

```
old_x, old_y = borne_inf, f(borne_inf)
# position initiale
alpha = atan(old_y/old_x) + pi # sachant que old_x est négatif
t.left(alpha)
t.up()
t.forward((old_x**2+old_y**2)**0.5)
t.right(alpha)
old_alpha = 0
t.down()

# trace la fonction
for x in range(borne_inf+1, borne_sup+1):
    y = f(x)
    diff_x = x - old_x
    diff_y = y - old_y
    d = (diff_x**2 + diff_y**2)**0.5 # distance
    alpha = atan(diff_y / diff_x) # angle
    t.left(alpha - old_alpha)
    t.forward(d)
    old_x, old_y, old_alpha = x, y, alpha

f = lambda x: eval(sys.argv[1]) # fonction à dessiner
turtle.reset() # initialise turtle
dessin_fun(turtle,f) # appel la fonction qui dessine
input("type return to finish ")
```

Un code plus élaboré est disponible ici. Il est utilisé avec la commande

```
python3 dessin_f.py f bi bs zoom
```

où

- `f` est la fonction à tracer
- `bi` la borne inférieure de l'intervalle où `f` est tracée
- `bs` la borne supérieure de l'intervalle
- `zoom` : le zoom pour le dessin

`dessin_f.py` trace les axes pour les valeurs -200..200 et utilise un incrément de 0.1 pour `x`. Pour cela, la fonction `range` est remplacée par un générateur `drange` défini dans le code. Un générateur est une sorte de fonction qui génère une séquence: pour cela il remplace le `return` habituel par la commande `yield` pour fournir les valeurs successives.

Note: les bibliothèques `pylab`, `numpy`, `scipy` et `matplotlib` permettent (!! en python 2.7 !!) de faire des calculs numériques et des dessins en particulier en utilisant de nombreuses fonctions disponibles.

6.6 Tableaux à plusieurs dimensions

6.6.1 Produit de 2 matrices $c = a * b$

Une *matrice* est représentée en *Python* sous forme de liste de listes de valeurs (exemple: $x = [[1, 2, 3], [4, 5, 6]]$ donne la matrice 2×3 x).

Le code suivant définit une fonction réalisant le produit de deux matrices données en paramètre et renvoie la matrice résultat.

```
def produit(a, b):
    """
        Produit de la matrice a par la matrice b
        Hypothèse a: m x l - b : l x n (produit possible)
    """
    c = []
    for i in range(len(a)):
        c_i = []
        for j in range(len(b[0])):
            c_i_j = 0
            for k in range(len(a[0])):
                c_i_j += a[i][k] * b[k][j]
            c_i.append(c_i_j)
        c.append(c_i)
    return c
```

```
x=[[1,2,3],[4,5,6]]
y=[[1,2],[4,5],[7,0]]
```

```
print("le produit vaut : ", produit(x,y))
```

Une version plus simple, crée la matrice résultat avec des valeurs 0 au départ et ensuite calcule les bonnes valeurs:

```
def produit(a, b):
    """
        Produit de la matrice a par la matrice b
        Hypothèse a: m x l - b : l x n (produit possible)
    """
    c = [[0 for i in range(len(b[0]))] for j in range(len(a))]
    for i in range(len(a)):
        for j in range(len(b[0])):
            for k in range(len(a[0])):
                c[i][j] += a[i][k] * b[k][j]
    return c
```

```
x=[[1,2,3],[4,5,6]]
y=[[1,2],[4,5],[7,0]]
```

```
print("le produit vaut : ", produit(x,y))
```

ENSEMBLES ET DICTIONNAIRE

See Also:

Lire le chapitre 10 du livre de Gérard Swinnen sur Python 3

7.1 Set

Le type ensemble (set) existe en *Python* y compris de nombreux opérateurs ensemblistes (exemple venant du tutoriel de python.org):

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket           # teste d'appartenance
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                               # lettres dans a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                           # lettres dans a mais pas dans b
{'r', 'd', 'b'}
>>> a | b                           # lettres soit dans a soit dans b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                           # lettres à la fois dans a et dans b
{'a', 'c'}
>>> a ^ b # lettres dans un des 2 mais pas dans les deux ensembles a et b
{'r', 'd', 'b', 'm', 'z', 'l'}
>>> s2 = set({})                   # ensemble vide : {} donne un dictionnaire vide
```

L'exemple précédent qui détermine les lettres communes de deux mots peut simplement s'écrire :

```
>>> s = set("pommeee")
>>> t = set("banane")
>>> s & t      # ensemble des lettres communes
{'e'}
>>> list(s & t) # si on veut une liste pas un ensemble
['e']
```

En une ligne :

```
>>> list(set("pommeee") & set("banane")) # liste des lettres communes
['e']
```

7.2 Dictionnaire

7.2.1 Définition et manipulation

Un dictionnaire est une séquence modifiable, où les indices peuvent être de (presque) n'importe quel type non modifiable.

Un dictionnaire peut être vu comme une correspondance entre un ensemble d'indices (les clés) et un ensemble de valeurs. À chaque clé correspond une valeur; l'association entre une clé et une valeur est vue comme une paire clé-valeur ou comme un élément de la séquence.

Exemple: construisons un dictionnaire anglais-français : les clés et les valeurs seront des chaînes de caractères.

```
>>> eng2fr = {'one': 'un', 'two' : 'deux', 'three' : 'trois'} # initialisation
>>> type(eng2fr)
<type 'dict'>
>>> eng2fr['four'] = 'quatre' # ajoute un élément d'"indice" 'four'
>>> print(eng2fr)
{'four': 'quatre', 'three': 'trois', 'two': 'deux', 'one': 'un'}
>>> print(eng2fr['two'])
deux
>>> print(eng2fr['five']) # élément inexistant
KeyError: 'five'
>>> dico2 = {} # dictionnaire vide
```

En général, l'ordre des éléments d'un dictionnaire est imprévisible. Ce n'est pas un problème : on n'accède pas aux éléments avec des indices entiers, on utilise les clés.

Type des valeurs et des clés: les valeurs peuvent être de n'importe quel type (comme pour les listes); les clés doivent être des éléments non modifiables (exemple: valeur simple, string, tuple) et donc pas une liste ou un autre dictionnaire qui sont des types modifiables.

```
>>> dico = {'bon': 'good', 'jour': 'day'}
>>> dico
{'jour': 'day', 'bon': 'good'}
>>> dico[(1,2)] = 36 # rajoute l'élément d'"indice" (1,2)
>>> dico[[3,4]] = 49 # erreur : car [3,4] est une liste : et ne peut être une clé d'un dictionnaire
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Plus précisément, le type d'une clé doit être "hachable", c-à-d que l'on peut le convertir en un nombre entier d'une manière déterministe.

```
>>> hash(6)
6
>>> hash('hello')
-1267296259
>>> hash((1, 2))
1299869600
```

Les objets modifiables ne possèdent pas cette propriété.

```
>>> hash([3, 4])
TypeError: unhashable type: 'list'
>>> hash({'bon': 'good'})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
>>> hash((1, 2, [3, 4]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Exemple plus compliqué:

```
>>> eng2fr = {'one': 'un', 'two': 'deux', 'three': 'trois'}
>>> d = {'entier': 1, 'liste': [1, 2, 3], 3: 'trois', 'dico': eng2fr}
>>> d
{'liste': [1, 2, 3], 'dico': {'three': 'trois', 'two': 'deux', 'one': 'un'}, 3: 'trois', 'entier': 1}
>>> d['entier']
1
>>> d['liste']
[1, 2, 3]
>>> d[3]
'trois'
>>> d['dico']['two']
'deux'
```

Un dictionnaire est une séquence : on peut lui appliquer certaines opérations déjà vues avec les autres séquences (strings, tuples, listes).

```
>>> d = {'entier': 1, 'liste': [1, 2, 3], 3: 'trois', 'dico': eng2fr}
>>> len(eng2fr)
4
>>> 'one' in eng2fr # eng2fr a une clé 'one'
True
>>> 'deux' in eng2fr # eng2fr n'a pas de clé 'deux'
False
>>> for elem in d: # pour toutes les clés du dictionnaire
```

```
...     print("l'élément : ", elem, " vaut ", d[elem])
...
l'élément :  liste  vaut  [1, 2, 3]
l'élément :  dico   vaut  {'three': 'trois', 'two': 'deux', 'one': 'un'}
l'élément :  3      vaut  trois
l'élément :  entier vaut  1
```

Par contre on ne pourra pas utiliser le “slice” puisqu’on ne travaille pas avec des indices entiers.

Il est fréquent d’utiliser des tuples comme clés d’un dictionnaire (principalement car on ne peut pas utiliser des listes). Par exemple, on peut utiliser un dictionnaire pour stocker un répertoire téléphonique dont les clés sont un tuple (nom, prénom).

```
>>> tel = {}
>>> tel['Baroud', 'Bill'] = '065-37-07-56'
>>> tel['II', 'Albert'] = '02-256-89-14'
>>> tel['Poelvoorde', 'Benoit'] = '081-23-89-65'
>>> for last, first in tel:
...     print('first, last, tel[last, first])
Benoit Poelvoorde 081-23-89-65
Bill Baroud 065-37-07-56
Albert II 02-256-89-14
```

Pour déterminer si une valeur est présente dans le dictionnaire, on peut utiliser la méthode `values` qui retourne toutes les valeurs sous la forme d’une liste.

```
>>> vals = eng2fr.values()
>>> 'deux' in vals
True
>>> print(vals)
['quatre', 'trois', 'deux', 'un']
```

7.2.2 Dictionnaires comme ensembles de compteurs

Problème (Histogramme de la fréquence des lettres): comment compter la fréquence de chaque lettre dans une chaîne de caractères ?

Plusieurs solutions :

- créer 26 variables, une pour chaque lettre. Traverser la chaîne et, pour chaque caractère, incrémenter le compteur correspondant, par ex. en utilisant une instruction conditionnelle chaînée (26 cas !).
- créer une liste de 26 éléments puis convertir chaque caractère en un nombre (en utilisant la fonction `ord()` par exemple). Utiliser ce nombre comme indice de la liste pour incrémenter le compteur approprié.
- créer un dictionnaire avec les caractères comme clés et les compteurs comme valeurs. La première fois que l’on rencontre un caractère, on l’ajoute dans le dictionnaire avec une valeur 1. Lors d’une prochaine occurrence de la lettre, on incrémente la valeur.

Avantage du dictionnaire : code simplifié et on utilise uniquement les compteurs utiles.

Solution utilisant un dictionnaire :

```
def histogram(s):
    d = {}
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d

>>> h = histogram('brontosaurus')
>>> print(h)
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

7.2.3 Méthodes des dictionnaires

Les objets de type dictionnaire possèdent une série de méthodes.

```
>>> dir(eng2fr)
['__class__', '__contains__', '__delattr__', '__delitem__', '__doc__',
...
'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values']
```

Nous allons en voir quelques unes.

La méthode `copy()` permet de faire une copie superficielle (shallow copy) d'un dictionnaire (qui n'est pas un alias).

La méthode `clear()` permet de supprimer tous les éléments du dictionnaire.

```
>>> d = {'yes' : 'oui', 'no' : 'non'}
>>> e = d.copy()
>>> e
{'yes': 'oui', 'no': 'non'}
>>> e['yes'] = 'Oui'
>>> e
{'yes': 'Oui', 'no': 'non'}
>>> d
{'yes': 'oui', 'no': 'non'}
>>> d.clear()
>>> d
{}
```

La méthode `get()` prend une clé et une valeur par défaut en paramètre. Si la clé est présente dans le dictionnaire, `get` retourne la valeur correspondante. Sinon, elle retourne la valeur par défaut.

```
>>> help(dict.get)
Help on method_descriptor:
```

```
get(...)
    D.get(k[,d]) -> D[k] if k in D, else d.  d defaults to None.
(END)
```

```
>>> d = {'yes' : 'oui', 'no' : 'non'}
>>> d.get('yes','Mot inconnu')
'oui'
>>> d.get('five','Mot inconnu')
'Mot inconnu'
```

Ceci permet de réécrire `histogram()` sans instruction conditionnelle explicite.

```
def histogram(s):
    d = {}
    for c in s:
        d[c] = d.get(c,0) + 1
    return d
```

La méthode `setdefault()` prend une clé et une valeur par défaut en paramètre. Si la clé existe : retourne la valeur, si la clé n'existe pas : ajoute la paire (clé, valeur par défaut) et retourne la valeur par défaut.

```
>>> eng2fr = {'one': 'un', 'two' : 'deux', 'three' : 'trois'}
>>> eng2fr.setdefault('two','dos')
'Deux'
>>> eng2fr.setdefault('ten', 'dix')
'dix'
>>> eng2fr
{'three': 'trois', 'two': 'deux', 'ten': 'dix', 'one': 'un'}
>>> 'one' in eng2fr
True
>>> 'five' in eng2fr
False
```

Comme l'exemple précédent l'illustre, l'opération `in` permet de savoir si une clé est présente dans le dictionnaire.

Les méthodes `keys`, `values` et `items` permettent d'obtenir respectivement les clés, les valeurs et des 2-uples "clé-valeur".

```
>>> eng2fr = {'one': 'un', 'two' : 'deux', 'three' : 'trois', 'four': 'quatre'}
>>> eng2fr.keys()
dict_keys(['four', 'three', 'two', 'one'])
>>> eng2fr.values()
dict_values(['quatre', 'trois', 'deux', 'un'])
>>> eng2fr.items()
dict_items([('four', 'quatre'), ('three', 'trois'), ('two', 'deux'), ('one', 'un')])
```

La méthode `pop` supprime une paire clé-valeur à partir d'une clé et retourne la valeur supprimée.

```
>>> eng2fr = {'one': 'un', 'two' : 'deux', 'three' : 'trois', 'four': 'quatre'}
>>> help(dict.pop)
Help on method_descriptor:
```



```
pop(...)
    D.pop(k[,d]) -> v, remove specified key and return the corresponding value.
    If key is not found, d is returned if given, otherwise KeyError is raised

>>> print(eng2fr.pop('five', None))
None
>>> print(eng2fr.pop('one', None))
un
>>> eng2fr
{'four': 'quatre', 'three': 'trois', 'two': 'deux'}
```

La méthode `popitem` supprime une paire clé-valeur (indéterminée) et retourne la paire supprimée sous la forme d'un 2-uple.

```
>>> eng2fr = {'one': 'un', 'two': 'deux', 'three': 'trois', 'four': 'quatre'}
>>> help(dict.popitem)
Help on method_descriptor:
popitem(...)
    D.popitem() -> (k, v), remove and return some (key, value) pair as a 2-tuple;
    but raise KeyError if D is empty.

>>> eng2fr.popitem() ('four', 'quatre') >>> eng2fr {'three': 'trois', 'two': 'deux'}
```

7.2.4 Quelques algorithmes sur des dictionnaires

Problème: trouver la (les) clé(s) d'une valeur donnée ?

Idee : on va traverser linéairement le dictionnaire et retourner une liste avec les clés correspondant à une valeur donnée (les clés sont uniques, mais il peut y avoir une même valeur pour différentes clés).

```
def get_keys(d, value):
    t = []
    for k in d:
        if d[k] == value:
            t.append(k)
    return t

>>> d = histogram('evenement')
>>> d {'m': 1, 'n': 2, 'e': 4, 't': 1, 'v': 1}
>>> get_keys(d, 4)
['e']
>>> get_keys(d, 1)
['m', 't', 'v']
```

La fonction `histogram` est pratique mais on pourrait avoir besoin d'obtenir ces informations de manière inversée.

Exemple : Au lieu de `{ 'm': 1, 'n': 2, 'e': 4, 't': 1, 'v': 1 }`,
on aimerait avoir `{ 1 : ['m', 't', 'v'], 2 : ['n'], 4: ['e'] }`.

Problème: comment inverser les clés et les valeurs d'un dictionnaire ?

Idée : on va créer un nouveau dictionnaire qui contiendra des listes : pour chaque valeur du dictionnaire de départ, on ajoute une paire dans le nouveau dictionnaire si celle-ci n'est pas présente, sinon, on met à jour la paire.

```
def invert_dict(d):
    inv = {}
    for k in d:
        val = d[k]
        if val not in inv:
            inv[val] = [k]
        else:
            inv[val].append(k)
    return inv

>>> d = histogram('evenement')
>>> inv_d = invert_dict(d)
>>> inv_d
{1: ['m', 't', 'v'], 2: ['n'], 4: ['e']}
```

7.3 Changement de type

Comme on l'a déjà vu *Python* a une batterie de fonctions prédéfinies pour passer d'un type à l'autre

Voici une liste plus complète :

- `int(x)` : donne la valeur entière tronquée de `x`
- `float(x)` : donne la valeur réelle (float) de `x`
- `bool(x)` : donne la valeur booléenne de `x` (si `x==0` : False, sinon True)
- `list(seq)` : transforme la séquence en liste
- `str(s)` : construit un string qui représente la valeur de `s`
- `repr(s)` : construit un string qui représente la valeur canonique de `s` (donc on utilise les notations `\t` quand il y a une tabulation `\n` pour le passage à la ligne ...)
- `tuple(s)` : transforme en tuple
- `dict(s)` : transforme en dictionnaire : `s` doit être une séquence de couples
- `set(s)` : transforme en ensemble

Notez que les fonctions qui génèrent des séquences peuvent parfois avoir des itérateurs comme argument (exemple: `set(range(10))`)

La méthode `dict()` utilise une liste de tuples pour initialiser un dictionnaire avec la fonction `dict`. Combiner `dict` et `zip` permet de créer de manière concise un dictionnaire.

```
>>> t = [('d', 3), ('e', 4), ('f', 5)]
>>> d = dict(t)
>>> print(d)
{'e': 4, 'd': 3, 'f': 5}
>>> d = dict(zip('abc', range(3)))
>>> print(d)
{'a': 0, 'c': 2, 'b': 1}
```

La méthode `update()` sur un dictionnaire prend une liste de tuples en argument et les ajoute au dictionnaire existant. En combinant items, l'assignation de tuples et une boucle `for`, on peut facilement traverser les clés et les valeurs d'un dictionnaire.

```
>>> d['a'] = '0'
>>> d.update(zip('bcd', range(1,4)))
>>> for key, val in d.items():
...     print(key, val)
a 0
c 2
b 1
d 3
```

7.3.1 Gather et scatter

Pour définir ses propres fonctions avec un nombre indéfini d'arguments, on utilise un paramètre dont le nom commence par `*`. Ce paramètre rassemble (gather) les arguments en un tuple, quel que soit leur nombre.

```
>>> def f(*args):
...     print(type(args), 'of length', len(args))
...     print(args)

>>> f(2, 4)
<type 'tuple'> of length 2 (2, 4)
>>> f('hello', 2.0, [1, 2])
<type 'tuple'> of length 3 ('hello', 2.0, [1, 2])
```

Remarque : il ne peut y avoir qu'un seul paramètre de ce type, et il doit se trouver à la fin de la liste des paramètres.

On peut combiner l'opérateur `gather` `*` avec des paramètres requis et optionnels (avec valeurs par défaut). On commence par les paramètres requis, puis les optionnels et enfin le paramètre de taille variable.

```
>>> def g(required, optional=0, *args):
...     print('required:', required)
...     print('optional:', optional)
...     print('others:', args)

>>> g()
TypeError: g() takes at least 1 argument (0 given)
```

```
>>> g(1)
required: 1
optional: 0
others: ()
>>> g(1,2)
required: 1
optional: 2
others: ()
>>> g(1,2,3)
required: 1
optional: 2
others: (3,)
>>> g(1,2,3,4)
required: 1
optional: 2
others: (3, 4)
```

L'opérateur `*` appliqué sur un tuple lors de l'appel à une fonction permet également de faire l'inverse : il sépare le tuple en une séquence d'arguments.

```
>>> help(divmod)
Help on built-in function divmod in module __builtin__:

divmod(...)
    divmod(x, y) -> (div, mod)

    Return the tuple ((x-x%y)/y, x%y). Invariant: div*y + mod == x.
>>> t = (7,3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
>>> divmod(*t)
(2, 1)
```

Warning: Nous avons vu que les séquences peuvent être manipulées soit avec des opérateurs (+, *, ...) soit avec des méthodes. En fait les opérateurs sont des notations alternatives de méthodes.

Par exemple si l'on demande la liste des attributs d'un entier (2 par exemple),

```
>>> dir(2)
['__abs__', '__add__', '__and__', '__bool__',
 '__ceil__', '__class__', '__delattr__', '__divmod__', '__doc__',
 '__eq__', '__float__', '__floor__', '__floordiv__', '__format__',
 '__ge__', '__getattribute__', '__getnewargs__', '__gt__',
 '__hash__', '__index__', '__init__', '__int__', '__invert__',
 '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__',
 '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__',
 '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
 '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__',
 '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__',
 '__rshift__', '__rsub__', '__rtruediv__', '__rxor__',
 '__setattr__', '__sizeof__', '__str__', '__sub__',
 '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag',
 'numerator', 'real', 'to_bytes']
```

la méthode `__add__` apparaît : l'opération + renvoie à cette méthode qui s'exécute donc lors de l'"appel" à l'opération.

Pour ne pas écrire de code erroné, il est, en particulier, essentiel de bien connaître l'effet d'une méthode: si l'objet est non modifiable (valeur simple, string, tuple, ...), généralement, la méthode renvoie un résultat (différent de `None`); dans le cas des objets modifiables (list, dict, set, ...), il faut bien voir, si l'effet de la méthode est

- de modifier l'objet mais renvoie `None`
- de renvoyer un résultat sans modifier l'objet
- à la fois de modifier l'objet et de renvoyer un résultat.

RECHERCHES ET TRIS

Un certain nombre d’algorithmes font partie de la culture informatique. Parmi ceux-ci, les algorithmes de

- recherche de l’élément minimum (ou maximum) dans une séquence;
- recherche d’un élément dans un ensemble représenté sous une forme ou une autre (séquence, séquence triée, structure plus élaborée);
- tri des éléments d’une séquence

sont probablement les “classiques des classiques”. Je me dois donc d’en donner une version Python.

Heureusement pour le programmeur, mais malheureusement pour l’algorithmicien désirant utiliser ses connaissances, ces fonctions sont déjà implémentées en *Python*:

```
>>> texte = "voici mon texte qui contient le mot que je cherche"
>>> min(texte)  # l'espace est l'élément qui a le plus petit code dans ce texte
, ,
>>> if "mot" in texte:
...     print("mot est dans mon texte")
...
mot est dans mon texte
>>> copains = ["Michelle", "Marie", "Alain", "Sophie", "Didier", "Ariane", "Gilles",
... "Bernadette", "Philippe", "Anne-Françoise", "Pierre"]
>>> x = input("Dis-moi ton nom")
>>> if x in copains:
...     print("bonjour", x)
...
bonjour Gilles
>>> copains.sort()
>>> for c in copains:
...     print(c)
...
Alain
Anne-Françoise
Ariane
Bernadette
Didier
Gilles
```

Marie
Michelle
Philippe
Pierre
Sophie

Notons qu'une recherche du minimum ou un tri dépend d'un *ordre*.

Un *ordre* pour un ensemble S d'éléments est une relation R *réflexive*, *transitive* et *anti-symétrique*. (Dans certains ouvrages, un ordre est une relation irréflexive, transitive et anti-symétrique. Dans ce cas nous parlerons d'*ordre strict*). L'ordre R est *total* si pour toute paire (a,b) d'éléments de S , $a R b$ ou $b R a$. Sinon, on dit que l'ordre est partiel, ce qui signifie qu'il peut exister des éléments a,b tels que ni $a R b$ ni $b R a$ ne sont vérifiées.

Dans d'autres langages où sont disponibles le `if`, `while` et `for` avec un indice (`for i in range(len(s))`), mais où la recherche ou le tri ne sont pas fournis au départ, on a des fonctions qui ressemblent aux codes suivants (à la syntaxe près). On suppose que les algorithmes de recherche ou de tri se passent sur des séquences (des listes pour les tris) dont les éléments sont des tuples (*clés*, *information satellite*).

Exemple: un annuaire: chaque *clé* est donnée par les prénom et nom d'une personne et l'information satellite est son adresse.

Les recherches et tris se font à partir des clés (recherche un élément ayant cette clé ou trie selon les clés) mais en cas de tri, tout l'élément y compris l'information satellite doit être déplacé. On suppose que chaque élément $s[i]$ de la liste s contient la clé en $s[i][0]$ (et donc l'information satellite en $s[i][1]$)

8.1 Les classiques : recherches

8.1.1 Recherche de l'élément minimum dans une séquence

Le principe est de retenir l'indice du meilleur candidat trouvé jusqu'à présent (0 au départ) et de parcourir la séquence complètement en comparant avec le candidat actuel.

```
def indice_min(s):  
    res = 0  
    for i in range(1, len(s)):  
        if s[i][0] < s[res][0]:  
            res = i  
    return res
```

Notons qu'ici c'est l'indice du minimum qui est renvoyé: cela permet d'avoir à la fois sa valeur $s[indice_min(s)]$ mais aussi sa position.

8.1.2 Recherche séquentielle

Le principe est de parcourir séquentiellement la séquence jusqu'à avoir trouvé ou être arrivé en fin de séquence. Notez l'utilisation de l'évaluation paresseuse avec l'opérateur `and`.

De nouveau, c'est l'indice dans `s` où se trouve la première instance de `x` qui est retournée.

```
def recherche(s, x):
    """ donne l'indice où se trouve la valeur de x dans s (-1 si n'existe pas) """
    i = 0
    while i < len(s) and s[i][0] != x:
        i = i+1
    if i == len(s):
        i = -1 # pas trouvé
    return i
```

Version améliorée

Si `s` est modifiable on peut ajouter l'élément recherché (et ensuite le retirer à la fin).

```
def recherche(s, x):
    """
        donne l'indice où se trouve la valeur de x dans s (-1 si n'existe pas)
        version améliorée quand s est modifiable
    """
    i = 0
    s.append((x, 0))
    while s[i][0] != x: # ne teste pas si on est en fin de séquence
        i = i+1
    del s[-1]
    if i == len(s):
        i = -1 # pas trouvé
    return i
```

8.1.3 Recherche dichotomique (ou binaire)

Cet algorithme effectue la recherche de la position d'un élément, de clé `x` donnée, dans une liste à `n` éléments indicés `0..n - 1` ; mais ici la liste est obligatoirement triée en ordre non décroissant sur les clés, ce qui veut dire que pour tout indice `i` et `j` de la liste avec `i <= j` on a sûrement `s[i][0] <= s[j][0]`.

Définissons nous une fonction *recherche_dichotomique* qui reçoit la liste `s` et la clé `x` et dont le résultat est l'indice, dans `s`, d'un élément quelconque dont la clé vaut `x` (ou `-1` si `x` n'est pas dans `s`). Sachant que `x` est supposé être dans la liste `s`, la recherche s'effectue dans la tranche de liste `s[bi:bs]`. Initialement `bi = 0` et `bs = len(s)`.

La recherche de `x` dans `s[bi:bs]` consiste à regarder l'élément milieu d'indice `m = (bi + bs)//2`.

1. Soit `s[m][0] < x` et il faut recommencer la recherche avec une nouvelle borne inférieure (`bi = m + 1`).

2. Soit $s[m][0] > x$ et il faut recommencer la recherche avec une nouvelle borne supérieure ($bs = m - 1$).
3. Soit $s[m][0] = x$ et m est l'indice recherché.

Cette recherche se termine soit lorsque $s[m][0] = x$ soit lorsque la tranche $[bi:bs]$ est vide. Dans le premier cas m est l'indice recherché sinon, par convention, le résultat de la recherche vaudra -1.

```
def recherche_dichotomique(s, x):  
    """  
        recherche dichotomique ou binaire de x dans s  
        résultat:  
            donne l'indice où se trouve la valeur de x dans s  
            (-1 si n'existe pas)  
    """  
    bi, bs = 0, len(s)  
    m = (bi+bs)//2  
    while bi < bs and x != s[m][0]:  
        m = (bi+bs)//2  
        if s[m][0] < x:  
            bi = m+1  
        else:  
            bs = m # x est avant ou est trouvé  
  
    if len(s) <= m or s[m][0] != x: # pas trouvé  
        m = -1  
    return m
```

8.2 Les classiques : tris

Trier une collection d'objets est l'opération qui consiste à les ranger selon l'ordre donné. Un tel rangement revient à effectuer une permutation des éléments pour les ordonner. Généralement la relation \leq est utilisée comme relation d'ordre.

Le tri peut être effectué

- soit par ordre croissant (ou plus précisément, non décroissant), ce qui signifie qu'on aura pour tout couple d'indice i, j avec $i < j$: $s[i] \leq s[j]$
- soit par ordre décroissant (ou plus précisément, non croissant), ce qui signifie qu'on aura pour tout couple d'indice i, j avec $i < j$: $s[i] \geq s[j]$.

Ici aussi on considère que les algorithmes de tri manipulent des listes d'éléments chacun contenant:

- une clé,
- une information satellite.

Par exemple, on pourrait avoir besoin de trier un tableau contenant des informations sur des personnes:

- les noms et prénoms peuvent constituer la clé sur laquelle on effectue les comparaisons pour trier les valeurs,
- les autres informations constituent l'information satellite associée.

Ainsi, après le tri, le tableau contiendra les informations sur les personnes, classées dans l'ordre alphabétique de leur nom. La figure suivante illustre un tel tableau avant et après le tri.

Avant le tri

Nom	Adresse
Vanbegin Marc	Av. Louise
Dupont Jean	rue du Moulin
Pascal Paul	...
Van Aa Michel	...
Dupont Jean	rue du Château
Milcamp René	rue de Liège
Alexandre Eric	...

Après le tri

Nom	Adresse
Alexandre Eric	...
Dupont Jean	rue du Moulin
Dupont Jean	rue du Château
Milcamp René	rue de Liège
Pascal Paul	...
Van Aa Michel	...
Vanbegin Marc	Av. Louise

Plusieurs remarques peuvent être formulées à partir de l'exemple précédent.

Tout d'abord, la relation d'ordre doit être précisément définie. Ici nous avons ignoré les espaces éventuels et considéré que les lettres majuscules sont équivalentes aux minuscules.

Ainsi: Van Aa = vanaa < vanbegin = Vanbegin.

Nous pouvons également constater que 2 Dupont Jean se trouvent dans la table; l'algorithme de tri peut, a priori, les classer dans n'importe quel ordre. Dans le cas où la table contient 2 entrées de même clé, il est parfois important que l'algorithme de tri conserve l'ancien ordre relatif: on dira alors que la méthode de tri est stable. On avait

- en indice 2 <Dupont Jean, rue du Moulin>
- en indice 5 <Dupont Jean, rue du Château>

et donc, si l'algorithme est stable, à la fin, <Dupont Jean, rue du Moulin> doit rester avant <Dupont Jean, rue du Château>

Il existe de nombreuses méthodes de tri. Dans les sections suivantes nous donnerons 4 méthodes de tri; chacune d'entre elles sera un représentant simple d'une technique de tri classiquement utilisée.

D'autres méthodes de tri ne seront pas présentées ici car elles font appel à des techniques de programmation non encore étudiées jusqu'à présent.

Comme pour les algorithmes de recherche, nous allons présenter nos algorithmes de tri sur des listes s d'éléments (les valeurs de $s[i][0]$ représentant les clés).

Notons également que les tris présentés ci-dessous rangent les éléments par *ordre non décroissant*.

8.2.1 Tri par sélection

Le tri par sélection présenté ici est l'exemple le plus simple des tris qui utilisent la technique consistant à sélectionner les éléments à trier dans un certain ordre, afin de les mettre, un par un, à leur place définitive.

À chaque étape, la sélection consiste à repérer un élément minimum parmi les éléments qui n'ont pas encore été correctement placés, et à le placer à sa position définitive.

Si parmi les éléments non encore placés correctement, il existe plusieurs valeurs minimales, la sélection choisit le premier, c'est-à-dire celui qui a un indice minimum, afin que le tri soit stable.

Ainsi, à l'étape 0, c'est-à-dire la première étape où l'on traite $s[0]$, la sélection choisit un minimum dans toute la liste (entre l'indice 0 et l'indice $n-1$ inclus), et l'échange avec $s[0]$.

À la deuxième étape le minimum sélectionné dans l'intervalle entre 1 et $n-1$ de s est échangé avec $s[1]$.

Ainsi, *au début* de l'étape i , les éléments $s[0]$ jusque $s[i-1]$ ont déjà pris leur place définitive, et les éléments $s[i]$ jusque $s[n-1]$ sont non encore triés **mais** on est assuré que toutes les valeurs de $s[i]$ jusque $s[n-1]$ sont supérieures ou égales à toutes les valeurs $s[0]$ jusque $s[i-1]$ incluses et en particulier à $s[i-1]$.

La situation de la liste au début de l'étape i du tri par sélection est illustré par la figure suivante :

éléments triés contenant les $i-1$ premiers minima	éléments non triés contenant des valeurs supérieures ou égales aux valeurs triées
composantes 0 à $i-1$	composantes i à $n-1$

Ainsi après l'étape $n-2$, la liste s est entièrement triée puisque $s[n-1]$ est supérieur ou égal aux éléments $s[0]$ jusque $s[n-2]$ triés.

La figure suivante montre les étapes du tri par sélection pour une liste dont on ne représente que les clés valant 1 à 7.

3	7	2	6	5	1	4
1	7	2	6	5	3	4
1	2	7	6	5	3	4
1	2	3	6	5	7	4
1	2	3	4	5	7	6
1	2	3	4	5	7	6
1	2	3	4	5	6	7

Code du tri par sélection d'une liste s .

```
def tri_selection(s):
    """
    Trie la liste s par sélection
    """
    n = len(s)
    for i in range(n-1):
        # recherche du min dans s[i..n]
        min = i # min = indice du min provisoire
        for j in range(i+1, n):
            if s[j][0] < s[min][0]:
                min = j
        # placement du ième élément: échange s[i] <-> s[min]
        s[min], s[i] = s[i], s[min]
```

Notons que ce tri n'est pas stable.

Essayons de nous assurer que l'algorithme fonctionne correctement même dans les cas limites.

- Si $n=0$ ou 1, l'algorithme ne fait rien.
- Sinon, si à l'étape i , le $i+1$ ème minimum est, au départ, en indice i , l'échange entre $s[i]$ et $s[\text{min}]$ ne modifie rien.

8.2.2 Tri par insertion

La technique consiste à considérer chaque élément à trier, un par un et à l'insérer en bonne place relative dans la partie des éléments déjà triés aux étapes précédentes.

À chaque étape, l'insertion d'un élément est effectuée.

Au départ, $s[0]$ est l'élément de référence.

À l'étape 1, l'élément $s[1]$ est placé correctement par rapport à la partie déjà triée, c'est-à-dire uniquement $s[0]$.

À la i ème étape, $s[i]$ est donc inséré.

La figure suivante donne la situation de s **au début** de cette i ème étape.

Situation de la liste au début de la i ème étape du tri par insertion

éléments triés entre eux	partie non encore modifiée
composantes 0 à $i-1$	composantes i à $n-1$

La i ème étape peut donc être décomposée en 3 parties:

- La recherche de la place j où doit s'insérer $s[i]$,
- Le déplacement vers la droite d'une position des éléments qui doivent se mettre après l'élément à insérer (car leur valeur est supérieure à celle de $s[i]$),
- L'insertion de $s[i]$ à sa nouvelle place.

Notons que la nouvelle place de la valeur insérée x n'est pas a priori, sa place définitive puisqu'il se peut que d'autres éléments soient insérés, par après, avant x .

La figure suivante montre les étapes du tri par insertion pour une liste dont on ne représente que les clés valant 1 à 7.

3	7	2	6	5	1	4
3	7	2	6	5	1	4
2	3	7	6	5	1	4
2	3	6	7	5	1	4
2	3	5	6	7	1	4
1	2	3	5	6	7	4
1	2	3	4	5	6	7

La procédure présentée à la figure suivante effectue le tri par insertion en fusionnant la partie 2 et 3. En effet, la recherche de la nouvelle position de l'élément à insérer (placé temporairement dans *Save* se fait dans ce cas ci linéairement en partant de l'indice $i-1$ et en descendant. Tant que $s[j]$ est strictement supérieur à l'élément à insérer, on le déplace en $s[j+1]$ et on décrémente j .

```
def tri_insertion(s):
    """
        Trie liste s par insertion
    """
    n = len(s)
    for i in range(1,n):
        Save = s[i]    #utilisé pour l'échange
        # insertion de s[i]
        j = i-1
        while j>=0 and s[j][0] > Save[0]:
            s[j+1] = s[j]
            j=j-1
        s[j+1] = Save
```

L'algorithme fonctionne même si $n=0$ ou 1 (dans ces cas il ne fait rien) et est stable.

8.2.3 Tri par échange (Bulle)

La technique d'échange consiste à comparer les éléments de la liste, 2 par 2, et à les permuter (ou échanger) s'ils ne sont pas dans le bon ordre, jusqu'à ce que la liste soit complètement triée.

Le tri Bulle effectue des comparaisons entre voisins.

À la première étape, $s[0]$ et $s[1]$ sont comparés et éventuellement permutés. Ensuite, l'algorithme compare $s[1]$ et $s[2]$, et ainsi de suite jusqu'à $s[n-2]$ et $s[n-1]$. Etant parti de $s[0]$ jusqu'à $s[n-2]$, on peut voir qu'après cette première étape, $s[n-1]$ contiendra le maximum des valeurs de s , la sous-liste de s d'indices $0..(n-2)$ contenant les autres valeurs déjà légèrement réorganisées.

À la 2ème étape, on recommence donc les comparaisons/échanges pour les valeurs entre $s[0]$ et $s[n-2]$, ce qui aura pour effet de mettre en place dans $s[n-2]$, le second maximum et de réorganiser à nouveau légèrement les autres valeurs dans la sous-liste de s d'indices $0..(n-3)$.

À la i ème étape, on devra donc réorganiser la sous-liste de s d'indices $0..(n-i)$.

La figure suivante montre les étapes du tri Bulle d'une liste dont on ne représente que les clés valant 1 à 7.

3	7	2	6	5	1	4
3	2	6	5	1	4	7
2	3	5	1	4	6	7
2	3	1	4	5	6	7
2	1	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7

Code du tri Bulle :

```
def tri_bulle(s):
    """
        Trie les n premières composantes de la liste s
        par méthode Bulle
    """
    n = len(s)
    for i in range(n, 1, -1): # réarrange s[0..i]
        for j in range(i-1):
            if s[j][0] > s[j+1][0]:
                s[j], s[j+1] = s[j+1], s[j]
```

Notons que l'algorithme donné donne un tri stable.

8.2.4 Tri par énumération (comptage)

La technique de tri par énumération consiste, grosso modo, à

- compter le nombre n_i de valeurs inférieures ou égales à toute valeur $s[i][0]$ de la liste à trier,
- sachant que n_i valeurs devront être placées avant la valeur $s[i]$, placer cette dernière à sa position définitive.

Pour que cette technique soit praticable, il faut:

- que l'intervalle des valeurs possibles dans s soit petit,
- que l'on puisse travailler avec une liste w de même type que s .

Si le nombre de valeurs différentes possibles est grand (fort supérieur à n), l'algorithme devient non efficace et prend trop de place mémoire.

Nous supposons ici que les valeurs possibles sont dans l'intervalle $[0..m-1]$. L'algorithme doit se réserver une liste *count* de m composantes entières (d'indice $0..m-1$), qui est utilisée pour effectuer le comptage des valeurs. m ne peut donc pas être trop grand.

L'algorithme effectue plusieurs tâches successives:

- le comptage dans la liste *count* du nombre de chacune des valeurs possibles (initialement `count[0] = -1`), les autres compteurs sont mis à 0; ceci permet à la fin de placer les éléments aux positions 0 à $n-1$.
- le calcul dans les différentes entrées *count[i]* du nombre de valeurs inférieures ou égales à i : cette étape revient à sommer dans *count[i]*, les valeurs de *count[0]* à *count[i]* calculées à l'étape 1.
- le placement dans w des éléments de s de façon triée. Cette étape se base sur le fait qu'après l'étape 2, si *count[i] = p*, un élément de valeur i pourra se mettre en position p dans w , le suivant en position $p-1$,
- le copiage de w dans s pour remettre le résultat dans la liste initiale.

L'algorithme est donné par le code:

```
m = 5
```

```
def tri_enumeration(s):  
    """  
        Trie les n premières composantes de la liste s  
        par énumération  
    """  
    n = len(s)  
    w = [0]*n  
    count = [-1]+[0]*(m-1)  
  
    # 1 Comptage  
    for j in range(n):  
        count[s[j][0]] +=1
```



```
# 2 Cumul des compteurs
for i in range(1,m):
    count[i] += count[i-1]

# 3 Placement des éléments dans w
for j in range(n-1,-1,-1):
    w[count[s[j]][0]] = s[j]
    count[s[j][0]] -= 1

# 4 Recopiage de w dans s
s[:] = w
```

Notons que:

- l'étape 3. procède par un **for** dont la variable de contrôle a sa valeur qui va en décroissant, ceci pour que le tri soit stable.

Si $m < n$ ce tri est très efficace (voir plus loin).

NOTION DE COMPLEXITÉ ET GRAND O

See Also:

Référence: livre de Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein - Algorithmique

Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein - Algorithmique - 3ème édition - Cours avec 957 exercices et 158 problèmes, Dunod, 2010, ISBN: 978-2-10-054526-1 <http://www.books-by-isbn.com/2-10/2100545264-Algorithmique-Cours-avec-957-exercices-et-158-problemes-2-10-054526-4.html>

9.1 Motivation

Montrer qu'une version d'un programme est plus *efficace* qu'une autre n'est, a priori, pas aisé. La succession ou l'imbrication des tests et des boucles et la multitude des possibilités fait qu'il n'est généralement pas possible ou raisonnable d'évaluer l'efficacité de l'algorithme pour chaque cas possible.

Dans ce chapitre, nous donnerons un aperçu des techniques et méthodes d'estimation de l'efficacité d'un algorithme. En particulier nous définirons la notation *grand O* qui permettra de classer les algorithmes selon leur type d'efficacité sans devoir détailler le nombre exact d'instructions exécutées par l'algorithme.

9.1.1 Critères de choix d'un algorithme

Généralement, il existe plusieurs méthodes pour résoudre un même problème. Il faut alors en choisir une et concevoir un algorithme pour cette méthode de telle sorte que cet algorithme satisfasse le mieux possible aux exigences.

Parmi les critères de sélection d'une méthode et en conséquence d'un algorithme, deux critères prédominent: la *simplicité* et l'*efficacité* de cet algorithme.

Si parfois ces critères vont de paire, bien souvent ils sont contradictoires; un algorithme efficace est, en effet, bien souvent compliqué et fait appel à des méthodes fort élaborées. Le concepteur

doit alors choisir entre la simplicité et l'efficacité.

Si l'algorithme devra être mis en oeuvre sur un nombre limité de données qui ne demanderont qu'un nombre limité de calculs, cet algorithme doit de préférence être simple. En effet un algorithme simple est plus facile à concevoir et a moins de chance d'être erroné que ne le sera un algorithme complexe.

Si, par contre, un algorithme est fréquemment exécuté pour une masse importante de données, il doit être le plus efficace possible.

Au lieu de parler de l'efficacité d'un algorithme, on parlera généralement de la notion opposée, à savoir, de la complexité d'un algorithme.

La complexité d'un algorithme ou d'un programme peut être mesurée de diverses façons. Généralement, le temps d'exécution est la mesure principale de la complexité d'un algorithme.

D'autres mesures sont possibles dont:

- la quantité d'espace mémoire occupée par le programme et en particulier par ses variables;
- la quantité d'espace disque nécessaire pour que le programme s'exécute;
- la quantité d'information qui doit être transférée (par lecture ou écriture) entre le programme et les disques ou entre le programme et des serveurs externes via un réseau;
- ...

Nous n'allons pas considérer de programme qui échange un grand nombre d'informations avec des disques ou un autre ordinateur.

En général nous analyserons le temps d'exécution d'un programme pour évaluer sa complexité. La mesure de l'espace mémoire occupé par les variables sera un autre critère qui pourra être utilisé ici.

De façon générale, la complexité d'un algorithme pour un ensemble de ressources donné est une mesure de la quantité de ces ressources utilisées par cet algorithme.

Le critère que nous utiliserons dans la suite est un nombre d'actions élémentaires exécutées. Nous prendrons soin, avant toute chose, de préciser le type d'actions prises en compte et si nous voulons une complexité minimale, moyenne ou maximale.

Nous verrons que, généralement, la complexité d'un algorithme est exprimée par une fonction qui dépend de la taille du problème à résoudre.

9.1.2 Exemple: la recherche du minimum

Le travail nécessaire pour qu'un programme s'exécute dépend de la 'taille' du jeu de données fourni.

Par exemple, si nous analysons la fonction `indice_min(s)` donnée au début du chapitre, (recherche l'indice de l'élément de valeur minimale dans une liste `s`, il est intuitivement normal que cet algorithme prenne un temps proportionnel à `len(s)`, et soit noté, par exemple, $T(\text{len}(s))$).

Dénotons $\text{len}(s) = n$. n est la taille de la liste et donc par extension, également la taille du problème.

Redonnons ici la partie traitement de cet algorithme en ayant eu soin de transformer le *for* en *while* pour mieux détailler chaque étape d'exécution de l'algorithme et nous supposons $n = \text{len}(s)$ connu dans le programme :

```

res = 0                # 1
i = 1                  # 2
while i < n :           # 3
    if s[i][0] < s[res][0]: # 4
        res = i          # 5
    i = i+1              # 6
return res              # 7

```

Pour obtenir des résultats qui restent valables quel que soit l'ordinateur utilisé, il faut réaliser des calculs simples donnant une approximation dans une unité qui soit indépendante du processeur utilisé. On peut, par exemple, faire l'approximation que chaque assignation de donnée simple ou test élémentaire représente une unité de temps d'exécution.

D'autres hypothèses auraient pu être faites. Par exemple, on aurait pu ne considérer que les assignations et rechercher la complexité en nombre d'assignations, ou ne considérer que les tests et rechercher la complexité en nombres de tests effectués par l'algorithme.

La complexité dépend donc fortement de l'unité prise qui doit être clairement précisée.

Avec nos hypothèses, les instructions des lignes 1, 2 et 7 prennent chacune une unité de temps d'exécution.

La boucle *while* de la ligne 3 à 6 s'exécute $n-1$ fois, mais le test s'effectue n fois. La ligne 3 prend n unités et la ligne 6, $n-1$ unités.

Le test de la ligne 4 prend une unité; il est effectué $n-1$ fois et donc la ligne 4 utilisera au total $n-1$ unités.

L'assignation de la ligne 5 prend une unité. Elle n'est effectuée que lorsque le test du *if* est vérifié. Si l'on ne désire pas uniquement faire une seule mesure sur un jeu de données bien précis, il faut donc expliciter si l'on désire connaître le temps d'exécution dans *le meilleur des cas* $T_{\min}(n)$, *le pire des cas* $T_{\max}(n)$, ou *en moyenne* $T_{\text{moyen}}(n)$. Ici:

- **$T_{\min}(n)$ est donné dans le cas où le test du *if* n'est jamais vérifié**, ce qui signifie que le minimum se trouve à la première composante. On a alors: $T_{\min}(n) = 1+1+n+(n-1)+(n-1)+1=3n+1$
- **$T_{\max}(n)$ est donné dans le cas où le test du *if* est à chaque fois vérifié**, ce qui correspond au cas où toutes les valeurs de s sont distinctes et triées en ordre décroissant. On a alors: $T_{\max}(n) = 1+1+n+(n-1)+(n-1)+(n-1)+1= 4n$

Le calcul de $T_{\text{moyen}}(n)$ est généralement beaucoup plus difficile à effectuer. Pour cela, on peut faire des hypothèses sur les jeux de données possibles et leur distribution statistique; ces hypothèses doivent être le plus réaliste possible. Pour que le calcul de $T_{\text{moyen}}(n)$ ne soit pas trop compliqué, on tolère généralement certaines hypothèses simplificatrices. Nous pouvons, par exemple, supposer que toutes les valeurs dans s sont distinctes et faire l'hypothèse que la distribution des valeurs est uniforme, la liste étant non triée a priori.

Pour faire ce calcul, séparons les actions dues au `while`, au `if` et à l'assignation finale, des actions d'assignation de la variable `res` et posons:

$C(n)$ = le nombre moyen d'assignations à `res` pour une liste de taille n

Si nous posons $R(n)$ = le nombre d'actions dues au `while`, au `if` et à l'assignation finale, ce nombre est connu et vaut: $R(n) = 3n$.

Nous avons: $T_{\text{moyen}}(n) = C(n) + R(n)$

Pour exprimer la valeur de $C(n)$, regardons d'abord sa valeur pour n valant 1, 2, ...

Pour n valant 1, $C(1)=1$ puisqu'il y a une assignation en début, et que le corps de la boucle `while` ne s'exécute jamais.

Pour n valant 2, la liste contient une valeur maximale *Max* et une valeur minimale *min*. Deux possibilités équiprobables peuvent se présenter:

1. $s = \text{min}$ suivi de *Max*
2. $s = \text{Max}$ suivi de *min*

Le cas 1. donne une assignation et le cas 2. deux assignations, soit en moyenne: $C(2) = 3/2$

Essayons d'avoir une expression générale pour $C(n)$: $s[0]$ a une chance sur n d'être le minimum, une chance sur n d'être le deuxième minimum, Donc $s[0]$ a $1/n$ chance d'être le i ème minimum et cela pour tout i entre 1 et n .

Si nous posons $C(0) = 0$ et si dans la sous-liste qu'il reste à traiter il y a j éléments plus petits que le minimum provisoire, le nombre d'assignations qu'il faudra encore effectuer vaudra, d'après la définition donnée, $C(j)$, car les nombres plus grands que le minimum provisoire n'impliqueront aucune assignation supplémentaire. Ce raisonnement est valable pour tout $i \geq 1$.

En conséquence:

$$C(i) = 1 + \frac{1}{i} \sum_{j=0}^{i-1} C(j) \quad (\forall i \geq 1)$$

le 1 étant dû à la première assignation (c'est-à-dire à l'instruction 1).

Pour avoir une idée précise de la valeur de $C(n)$, il faut éliminer les $C(j)$ du membre de droite.

Pour cela exprimons cette équation d'une autre façon en le multipliant par i

$$i.C(i) = i + \sum_{j=1}^{i-1} C(j)$$

et de même si on multiplie $C(i+1)$ par $i+1$:

$$(i + 1).C(i + 1) = (i + 1) + \sum_{j=1}^i C(j)$$

En soustrayant la seconde équation de la troisième nous obtenons:

$$(i + 1).C(i + 1) - i.C(i) = 1 + C(i)$$

Ce qui peut s'écrire:

$$C(i + 1) = \frac{1}{i + 1} + C(i) \quad (\forall i \geq 1)$$

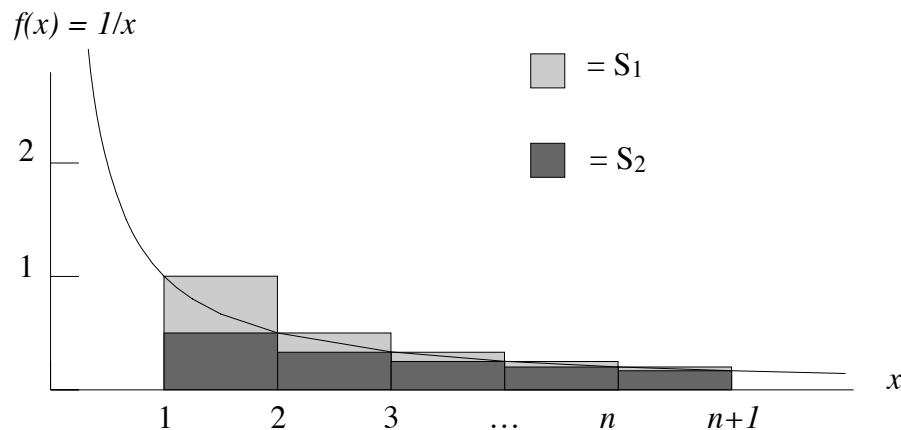
Pour $C(1)$ la formule reste vrai ($C(1)=1$)

Et donc:

$$C(n) = \frac{1}{n} + \frac{1}{n-1} + \dots + 1 = \sum_{i=1}^n \frac{1}{i}$$

Notons que nous aurions pu déduire ce résultat plus rapidement en exprimant que $C(i+1)$ vaut le nombre moyen d'assignations dû à la recherche du minimum pour les i premiers nombres, c'est-à-dire $C(i)$, plus la probabilité que le nombre $s[i]$ soit le minimum, c'est-à-dire $1/i+1$, ce qui nous redonne l'équation précédente.

Pour effectuer une approximation de $C(n)$, évaluons les aires $S1$ et $S2$ données en grisé dans la figure suivante où la fonction $f(x) = 1/x$.



Ayant

$$\int_1^{n+1} \frac{1}{x} dx = \ln(n+1)$$

on voit que

$$S2 < \ln(n+1) < S1$$

Comme

$$S1 = 1 + \frac{1}{2} + \dots + \frac{1}{n} = \sum_{i=1}^n \frac{1}{i} = C(n)$$

$$S2 = \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n+1} = \sum_{i=2}^{n+1} \frac{1}{i} = C(n+1) - 1$$

et ceci pour tout $n \geq 1$.

On a donc:

$$C(n+1) - 1 < \ln(n+1) < C(n) \quad (\forall n \geq 1)$$

De ce fait:

$$C(n) < \ln(n) + 1 \quad (\forall n > 1)$$

Et finalement, comme $\ln(n) < \ln(n+1)$, on obtient:

$$\ln(n) < C(n) < \ln(n) + 1$$

Cette dernière équation permet de borner $T_{moyen}(n)$ (rappelons-nous que $T_{moyen}(n) = C(n) + R(n)$ avec $R(n) = 3n$)

Nous obtenons donc:

$$3n + \ln(n) < T_{moyen}(n) < 3n + \ln(n) + 1$$

9.1.3 Autre exemple: la recherche d'un élément

Calculons les $T_{min}(n)$, $T_{Max}(n)$ et $T_{moyen}(n)$ de l'algorithme de recherche d'un élément dans une liste s .

Redonnons ici la partie traitement de cet algorithme en supposant que $n = \text{len}(s)$:

```

i = 0                                #1
while i < n and s[i][0] != x:         #2
    i = i+1                           #3
if i == n:                           #4
    i = -1 # pas trouvé               #5
return i                             #6

```

$T_{min}(n)$ est donné lorsque l'on trouve directement l'élément.

Si l'on suppose ici encore que toute assignation et tout test élémentaire prend une unité de temps et que le test du while est composé de deux tests élémentaires (le temps pour le and étant omis), on obtient: $T_{min}(n) = 5$ (c'est-à-dire 1 en #1 + 2 en #2 + 1 en #4 + 1 en #6)

$T_{MAX}(n)$ est donné dans le cas où l'élément x n'est pas dans s : $T_{MAX}(n) = 3n+5$

$T_{moyen}(n)$ dépend de la probabilité p que x soit dans s .

En supposant que si x est dans s , il a la même probabilité d'être dans n'importe quelle composante $s[i]$ (pour $0 \leq i < n$), en énumérant les cas possibles on obtient:

Cas possibles	Nombre d'unités
1: on trouve x en $s[0]$	$3+2$
2: on trouve x en $s[1]$	$6+2$
...	
n : on trouve x en $s[n-1]$	$3n+2$

Si x est dans s :

$$\begin{aligned}
 \text{moyen} &= \frac{3}{n}(1 + 2 + \dots + n) + 2 \\
 &= \frac{3(n+1)}{2} + 2 \quad \left(\text{puisque } \sum_{i=1}^n i = \frac{(1+n)n}{2}\right) \\
 &= \frac{3}{2}n + \frac{7}{2}
 \end{aligned}$$

Si x n'est pas dans s :

$$\min = \text{moyen} = \text{MAX} = 3n+5$$

Finalement

$$T_{moyen}(n) = (1 - p)(3n + 5) + p\left(\frac{3}{2}n + \frac{7}{2}\right) = \left(3 - \frac{3}{2}p\right)n + \left(-\frac{3}{2}p + 5\right)$$

9.2 Le grand O

Les exemples précédents montrent qu'il est souvent très difficile de calculer le temps moyen (ou maximal) d'exécution d'un algorithme. Ces calculs précis sont, de plus, inutiles puisqu'ils se basent sur des approximations grossières du temps d'exécution des actions élémentaires; évaluations qui ne tiennent pas non plus compte, par exemple, de l'ordinateur et du compilateur ou de l'interpréteur utilisés.

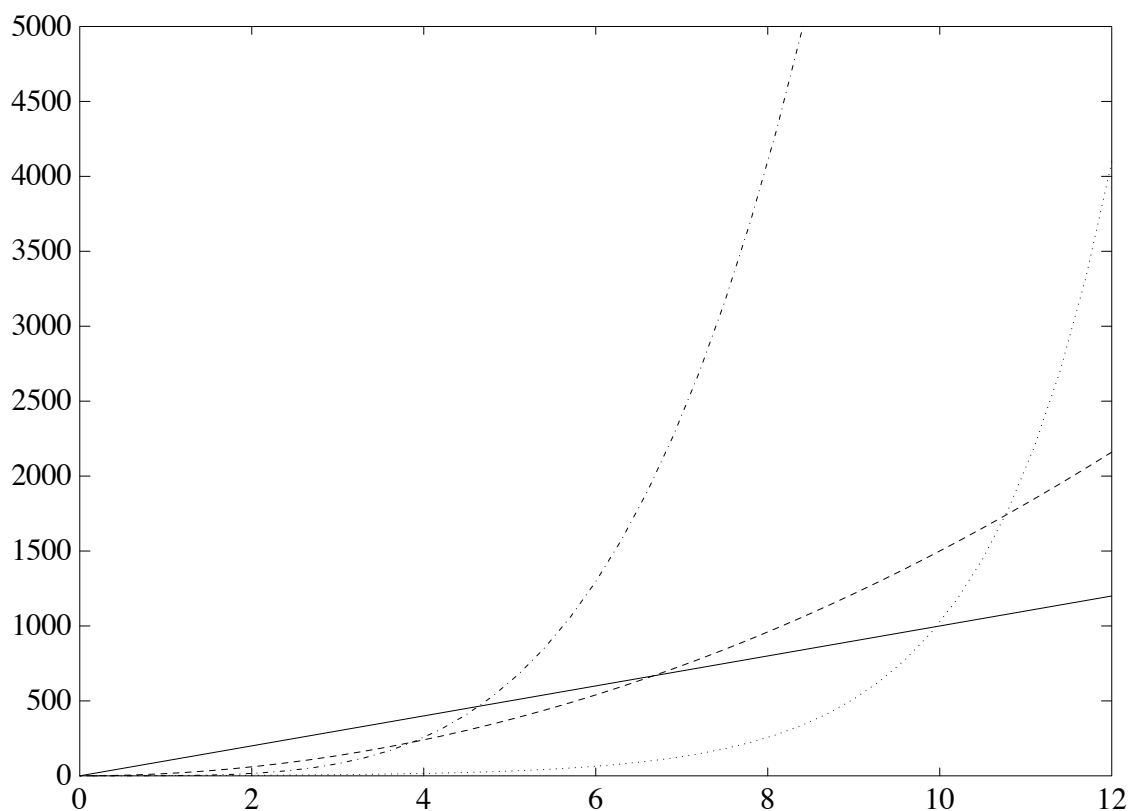
Une évaluation du temps d'exécution d'un algorithme ne devient généralement intéressante que lorsque la taille du jeu de données, c'est-à-dire de n dans nos exemples précédents, est grande. En effet pour n petit, les algorithmes s'exécutent généralement très rapidement.

Supposons que la complexité maximale d'un algorithme $A = 100.n$, que la complexité maximale d'un algorithme $B = 15.n^2$, celle de $C = n^4$ et celle de $D = 2^n$.

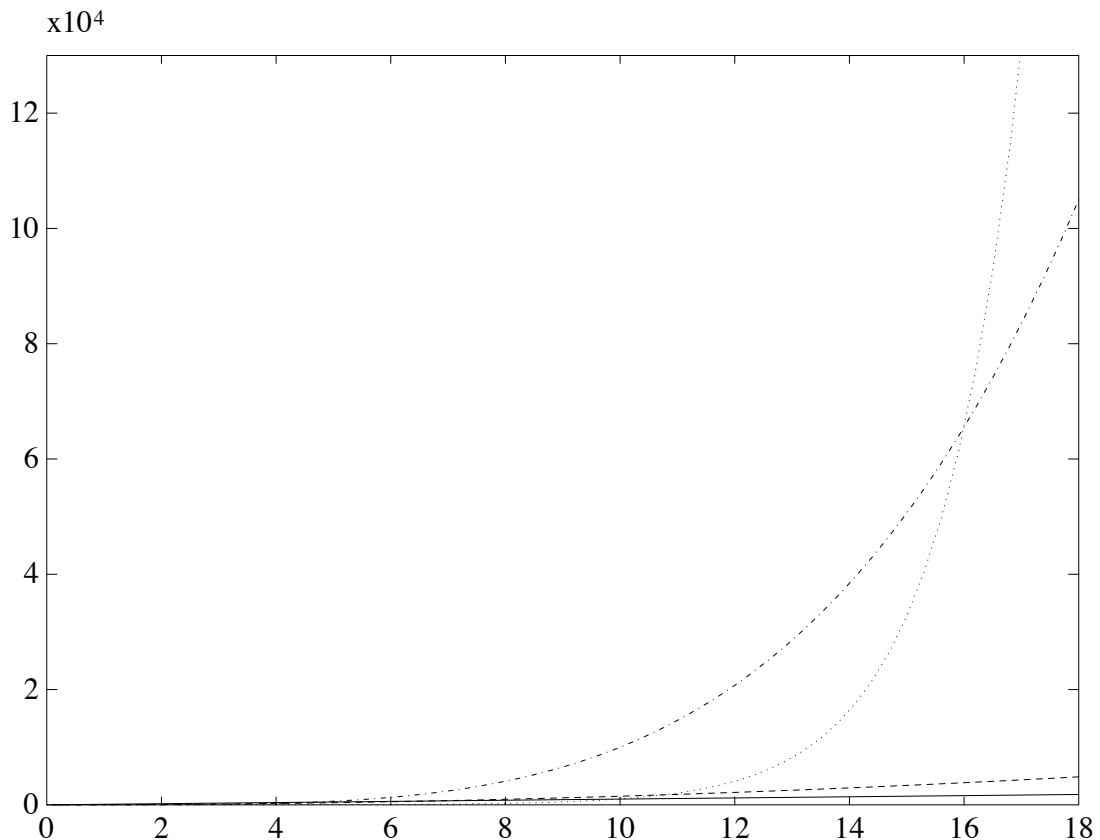
La table suivante donne les valeurs de complexité pour $n = 1, 10, 100, 1000, 10^6$ et 10^9 ; les figures qui suivent donnent les valeurs des fonctions respectivement pour n variant entre 0 et 12, et 0 et 18.

$n =$	A	B	C	D
1	100	15	1	2
10	1000	1500	10000	$\approx 10^3$
100	10000	150000	10^8	$\approx 10^{30}$
1000	100000	$15 \cdot 10^6$	10^{12}	$\approx 10^{300}$
10000	10^6	$15 \cdot 10^8$	10^{16}	$\approx 10^{3000}$
10^6	10^8	$15 \cdot 10^{12}$	10^{24}	$\approx 10^{300000}$
10^9	10^{11}	$15 \cdot 10^{18}$	10^{36}	$\approx 10^{3 \cdot 10^8}$

Valeur de $100.n$, $15.n^2$, n^4 et 2^n pour différentes valeurs de n



Valeur des fonctions $A = -$, $B = - -$, $C = - . -$ et $D = \dots$ pour n allant de 0 à 12



Valeur des fonctions $A = -$, $B = - -$, $C = -.-$ et $D = \dots$ pour n allant de 0 à 18

Ces valeurs doivent être divisées par le nombre d'instructions élémentaires par seconde pour obtenir un temps exprimé en seconde.

Les micro ou mini ordinateurs actuels permettent d'exécuter de l'ordre de 10^9 instructions par seconde soit environ 10^{14} instructions par jour.

Cela signifie qu'en un jour, si aucune autre limitation ne perturbe l'exécution des algorithmes (telle que l'espace mémoire disponible par exemple), on peut résoudre avec l'algorithme

- A , un problème pour n valant approximativement 10^{12}
- B , un problème pour n valant approximativement 10^7
- C , un problème pour n valant approximativement 3000
- D , un problème pour n valant approximativement 50

et donc A est meilleur que B , B est meilleur que C et C est meilleur que D (sauf pour de petites valeurs de n).

En vertu des remarques précédentes et de ces chiffres, on préfère donner un ordre de grandeur permettant d'avoir une idée du type d'algorithme. Pour cela, on utilise la notation grand O définie ci-dessous, qui permet:

- de déterminer si un algorithme a une chance de s'exécuter pour un n donné,

- connaissant le temps d'exécution pour un n donné, de faire une approximation de ce temps pour une autre valeur de n .

9.2.1 Définition

La notation grand O est définie comme suit:

Définition (O) : Une complexité $T(n)$ est dite en grand O de $f(n)$ (en $O(f(n))$) s'il existe un entier N et une constante $c > 0$ tels que pour tout entier $n > N$ nous avons $T(n) \leq c.f(n)$

Cette définition permet d'effectuer de nombreuses simplifications dans les calculs. En effet de cette définition il résulte que:

- Les facteurs constants ne sont pas importants. En effet, si par exemple $T(n)=n$ ou si $T(n)=100n$, $T(n)$ est toujours en $O(n)$
- Les termes d'ordres inférieurs sont négligeables. Ainsi si $T(n) = n^3 + 4n^2 + 20n + 100$, on peut voir que pour $n > N = 5$

$$- n^3 > 4n^2$$

$$- n^3 > 20n$$

$$- n^3 > 100$$

et donc pour $n > 5$, $T(n) = n^3 + 4n^2 + 20n + 100 < 4n^3$.

De ce fait $T(n)$ est en $O(n^3)$.

9.2.2 Calcul du grand O

Classes de complexité

Si on peut calculer qu'un algorithme est un $O(n^2)$, il n'est pas intéressant, même si cela est trivialement vrai, d'exprimer le fait que $T(n)$ est en $O(n^3)$, $O(n^4)$,

Lorsque l'on évalue le grand O d'une complexité, c'est la meilleure estimation "simple" que l'on cherche à obtenir.

Ainsi on classe généralement les complexités d'algorithmes selon leur grand O comme donné par la table suivante :

O	Classe d'algorithmes
$O(1)$	constant
$O(\log n)$	logarithmique
$O(n)$	linéaire
$O(n \log n)$	$n \log n$
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel en base 2
$O(3^n)$	exponentiel en base 3
...	
$O(n^n)$	exponentiel en base n
...	

Classes de complexité

Dans la suite, nous essayerons toujours de donner une approximation de la complexité des algorithmes vus.

Remarquons qu'en vertu du fait que *pour tout* a, b positif : $\log_a n = (\log_a b) \cdot (\log_b n)$

et comme $\log_a b$ est une constante pour a et b fixés, si $T(n)$ est en $O(\log_a n)$ il est également en $O(\log_b n)$. (la base du logarithme n'a donc pas d'importance pour exprimer un grand O). Notons encore que ceci n'est pas vrai pour la base des complexités exponentielles.

Donnons ici un petit nombre de règles permettant d'évaluer la complexité d'un algorithme. Ces règles ne donnent en général qu'une surapproximation parfois grossière; une estimation plus précise devant tenir compte du fonctionnement de l'algorithme.

Règles de calcul du grand O

Règle sur l'unité :

Règle 1 (unité)

Il faut clairement définir l'unité utilisée et les éléments pris en compte. Ainsi si l'on tient compte des tests élémentaires, des assignations, des lectures de données et des écritures de résultats, une assignation à une variable simple, une instruction input ou print d'une donnée simple, ou l'évaluation d'une expression (ou d'une condition) simple ne donnant pas lieu à l'exécution de fonctions, prend un temps constant fixé et est donc en $O(1)$. L'assignation à des objets plus complexes ou des input ou print de données plus complexes peut ne plus être en $O(1)$ en fonction de leur taille qui peut dépendre de la taille du problème (n).

Règle de la séquence :

Règle 2 (séquence)

Si un traitement 1 prend un temps $T_1(n)$ qui est en $O(f_1(n))$ et un traitement 2 prend un temps $T_2(n)$ qui est en $O(f_2(n))$ alors le traitement 1 suivi du traitement 2 prend $T_1(n) + T_2(n)$ et est en

$$O(f_1(n) + f_2(n)) = O(\max(f_1(n), f_2(n)))$$

où max prend la fonction qui croît le plus vite c'est-à-dire de plus grand "ordre".

Par exemple si $T_1(n)$ est en $O(n^2)$ et $T_2(n)$ est en $O(n)$, $T_1(n) + T_2(n)$ est en $O(n^2 + n) = O(n^2)$

En particulier une séquence d'instructions simples (dont on tient compte) ne faisant aucun appel à une procédure ou à une fonction ne dépend pas de n et sont donc en $O(1)$.

Règle du if :

Règle 3 (if)

Pour un `if condition: instruction_1 else: instruction_2`

où

- `instruction_1` est en $O(f_1(n))$
- `instruction_2` est en $O(f_2(n))$
- l'évaluation de la `condition` est en $O(g(n))$

suitant le test, le `if` sera en $O(\max(f_1(n), g(n)))$ ou en $O(\max(f_2(n), g(n)))$ et peut être borné par:

$$O(\max(f_1(n), f_2(n), g(n)))$$

Notons que souvent $g(n)$ est en $O(1)$.

Cette règle peut être généralisée pour une instruction `if` ayant une ou des parties `elsif`.

Règle du while :

Règle 4 (while)

Pour un `while`, sachant que le corps de la boucle est en $O(f_1(n))$ et que l'évaluation de la condition est en $O(f_2(n))$, si on a une fonction en $O(g(n))$ qui donne une borne supérieure du nombre de fois que le corps sera exécuté, alors le `while` est en $O(f(n).g(n))$ avec $f(n) = \max(f_1(n), f_2(n))$.

Règle du for :

Règle 5 (for)

Pour une boucle `for`, il suffit de “traduire” le `for` en `while`.

Par exemple :

```
for i in range(n):  
    print(i)
```

se traduit en:

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

La fonction `iter()` appliquée sur une séquence ou un range, permet de pouvoir ensuite utiliser la fonction `next()` sur l’objet produit.

Par exemple :

```
st = "bonjour"  
for i in st:  
    print(i)
```

se traduit en:

```
st = "bonjour"  
i = iter(st)  
s=next(i)  
while s != None:  
    print(s)  
    s=next(i)
```

Donnons de simples exemples de complexité avec des `for` où ici aussi n est la taille du problème :

```
for i in range(10):  
    traitement
```

est en $O(10.f(n))$ où $O(f(n))$ est la complexité d’une exécution du traitement et donc le code complet est également en $O(f(n))$

```
for i in range(n):  
    traitement
```

est en $O(n.f(n))$ où $O(f(n))$ est la complexité d’une exécution du traitement.

Ainsi le produit de matrice ($M.L$ par $L.N$) est en $O(M.L.N)$.

Règle de la fonction :

Règle 6 (fonction)

L'appel à une fonction est en $O(f(n))$ correspondant à la complexité du traitement de cette fonction pour les paramètres effectifs donnés.

On peut raffiner ces calculs, selon que l'on essaye de trouver une complexité minimale, moyenne ou maximale. Ce raffinement se situera dans le nombre de fois qu'une instruction sera répétée, ayant par exemple certaines hypothèses sur la probabilité que certaines conditions de `if` ou de boucles sont vérifiées.

9.3 Application des règles de calcul

Ayant ces règles d'évaluation du grand O pour chaque type d'instruction, le calcul de la complexité d'un algorithme se fait en partant des complexités des instructions simples, et en calculant, de proche en proche, les complexités des instructions non simples à partir des résultats déjà calculés. Ces calculs procèdent en quelque sorte par des regroupements en suivant la structure de l'algorithme.

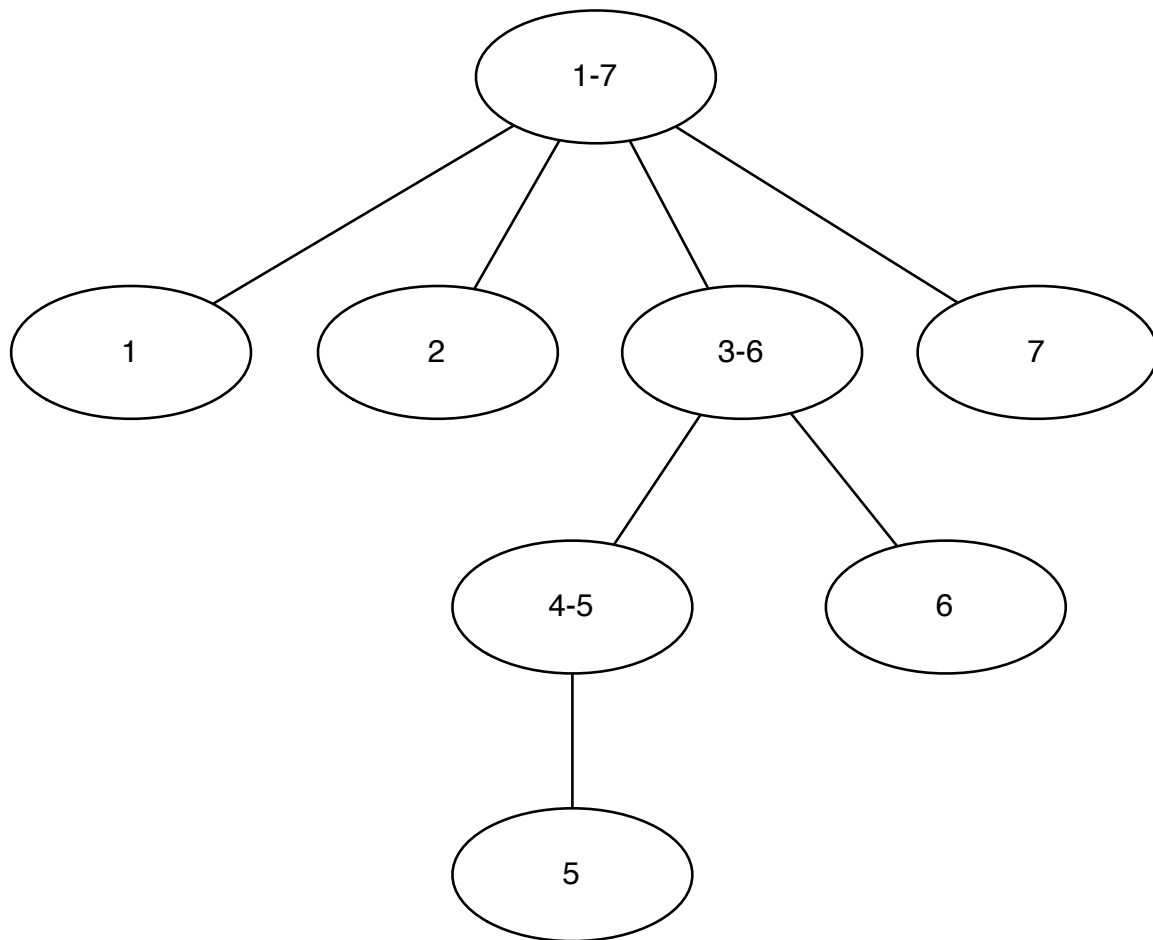
9.3.1 Complexité de la recherche du minimum

Prenons l'exemple de la recherche du minimum c'est-à-dire du traitement suivant (après traduction en utilisant un `while` en ayant `n = len(s)`):

```
res = 0                                #1
i = 1                                  #2
while i < n :                           #3
    if s[i][0] < s[res][0]:             #4
        res = i                        #5
    i = i+1                             #6
return res                             #7
```

Nous supposons que chaque instruction simple est prise en compte pour le calcul de la complexité.

Le traitement, comme tout algorithme, peut être décomposé sous forme d'*arbre (informatique)* comme donné par la figure suivante où chaque *noeud représente* une instruction *Python* simple ou composée.



Décomposition d'un traitement en suivant la structure

Les noeuds sans *fil*s, appelés les *feuilles* de l'arbre, correspondant aux instructions 1, 2, 5, 6 et 7, représentent des instructions d'assignations simples ou `return` qui, d'après la règle 1, sont en $O(1)$.

De ce fait, d'après la règle 3, le noeud 4-5 correspondant à l'instruction `if` est en $O(1)$.

D'après la règle 2, la complexité de la séquence d'instructions 4-6 est en $O(1)$.

On peut maintenant évaluer la complexité du `while` (noeud 3-6), il s'exécute $n-1$ fois et la complexité du corps de la boucle est, comme on vient de le voir, en $O(1)$. Donc d'après la règle 4, la complexité du `while` est en $O(n)$.

Finalement, d'après la règle 2, la complexité de tout le traitement est en $O(1+1+n+1)$ qui peut être simplifié par $O(n)$.

Note:

- Si, par exemple, un algorithme A est en $O(n)$ et un algorithme B est en $O(n^2)$, A est 'meilleur', en terme, de complexité que B .

Par contre si A et B sont tous les deux en $O(f(n))$, il faut détailler le calcul de la complexité pour déterminer lequel est plus complexe que l'autre.

- D'après la définition du O , l'algorithme de recherche du minimum est en $O(n)$ et donc trivialement également en $O(n^2)$, $O(n^3)$, ..., $O(n^n)$. Lorsque l'on cherche la complexité d'un algorithme, on essaye bien évidemment de donner la valeur la plus précise (Exemple: $O(n)$ pour la recherche du minimum)
 - Rappelons que l'on peut évaluer plusieurs complexités différentes. Par exemple, pour les algorithmes de tris de liste, on peut regarder la complexité en terme de nombre d'instructions ou seulement regarder les instructions qui effectuent des déplacements d'éléments (le déplacement d'un élément peut prendre beaucoup plus de temps que les autres instructions). Certains tris sont en $O(n^2)$ si l'on regarde toutes les instructions, mais en $O(n)$ en terme de déplacement d'informations.
-

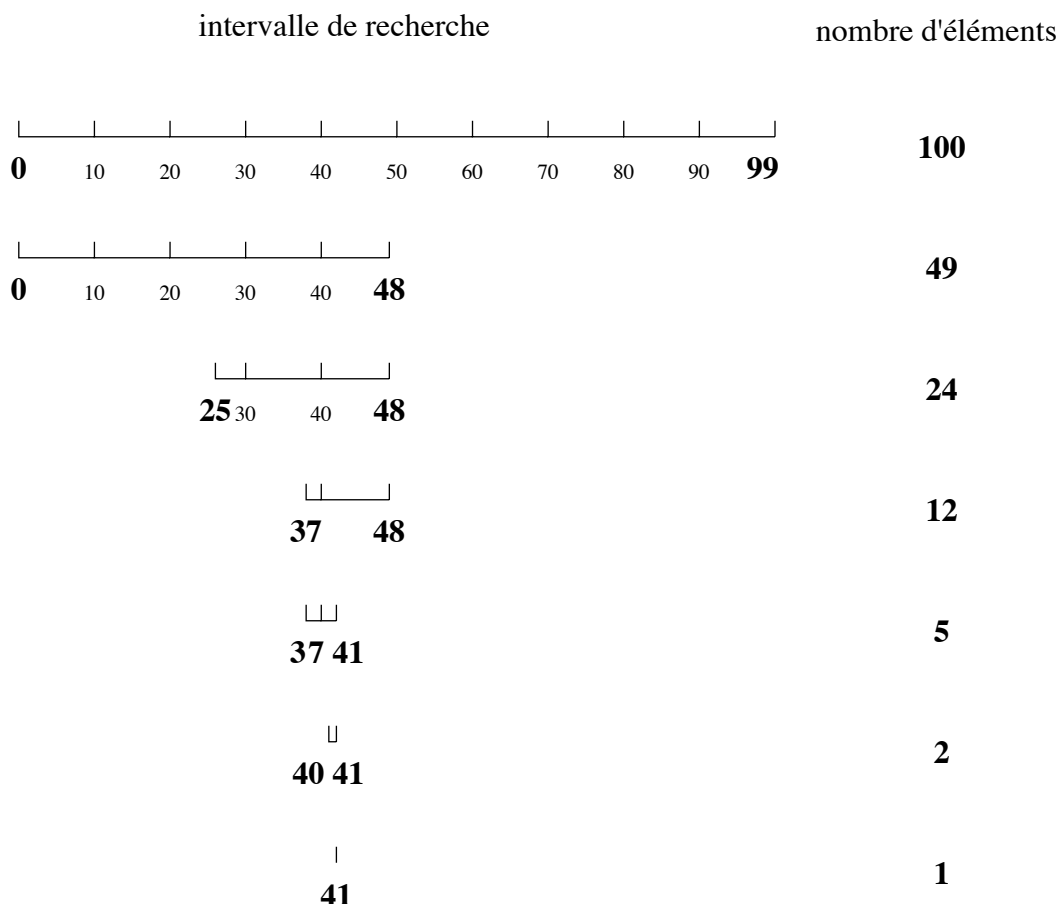
9.3.2 Complexité de la recherche séquentielle et dichotomique

Il est facile de voir que l'algorithme de **recherche séquentielle** est également en $O(n)$ où n est la longueur de la liste et en supposant que les éléments sont testés en $O(1)$.

Pour calculer la complexité de la **recherche dichotomique**, en utilisant les règles vues précédemment, on peut assez rapidement observer que la complexité moyenne ou maximale en nombre d'actions élémentaires de cet algorithme est en $O(f(n))$ où $f(n)$ est le nombre respectivement moyen ($f_{moyen}(n)$) ou maximum ($f_{max}(n)$) de fois que la boucle `while` est exécutée.

Pour évaluer $f_{max}(n)$, commençons par analyser le comportement de l'algorithme. Nous pouvons tout d'abord remarquer qu'à chaque tour de boucle, au pire des cas, l'intervalle de recherche est divisé par deux, quelle que soit la valeur de x .

La figure suivante donne une évolution possible de l'intervalle de recherche dans s pour une liste à 100 éléments en supposant que x n'est pas dans s mais que $s[40] < x < s[41]$



Evolution de l'intervalle de recherche lors d'une recherche dichotomique

Le `while` s'exécute donc au maximum 7 fois. De façon générale on peut voir que:

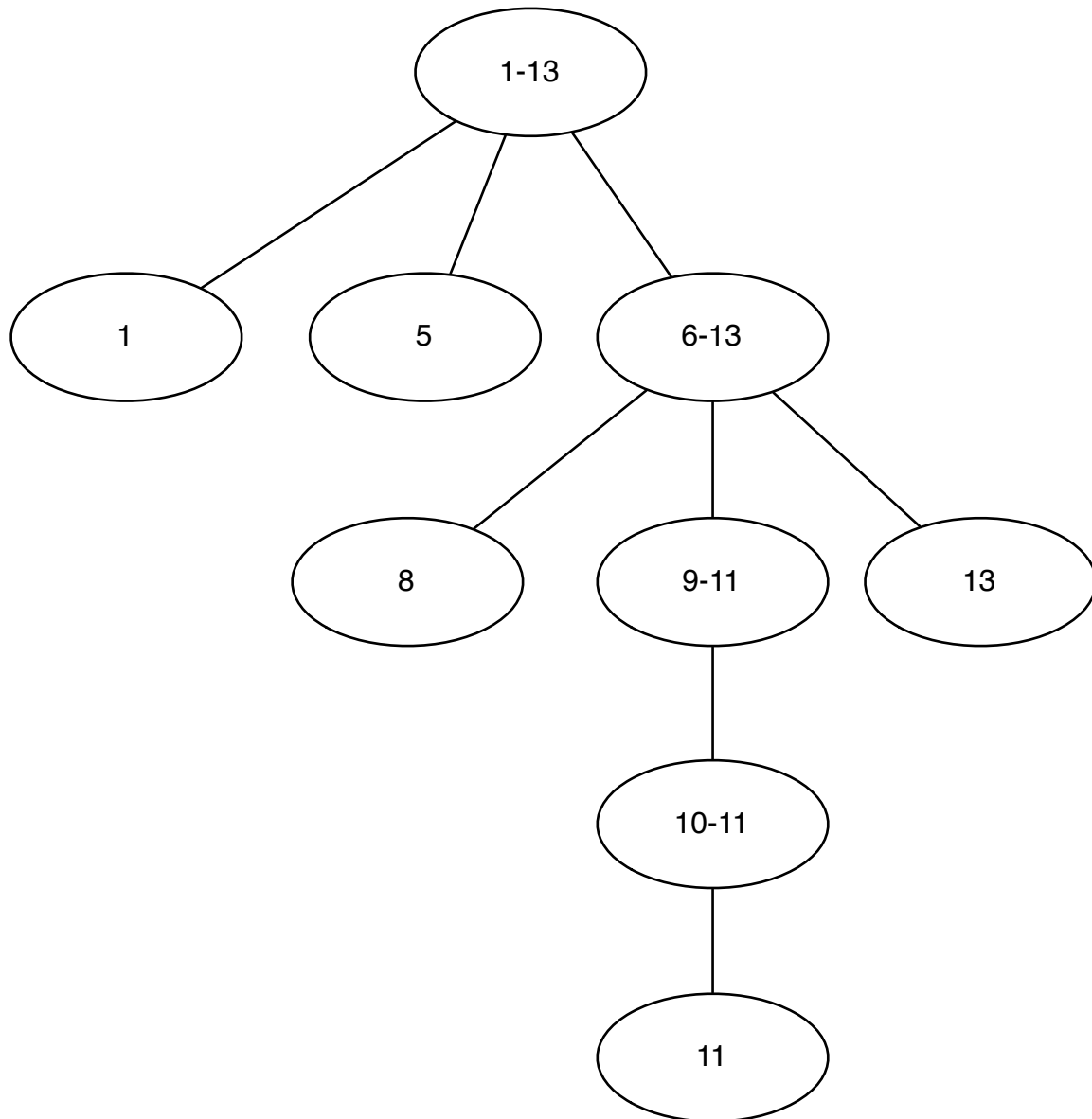
$$2^{f_{max}(n)-1} \leq n < 2^{f_{max}(n)}$$

et donc que $f_{max}(n) \leq \log_2(n) + 1$

La recherche dichotomique a donc une complexité maximale en $O(\ln n)$. On peut calculer que la complexité moyenne (en supposant ou non que x est dans s) est également en $O(\ln n)$.

9.3.3 Complexité du tri par sélection

Pour calculer la complexité maximale du tri par sélection, décomposons l'algorithme sous forme d'arbre en suivant sa structure; ceci est donné en figure suivante:



Décomposition du tri par sélection

En supposant que toutes les instructions et les traitements de passages de paramètres interviennent dans ce calcul, en suivant les règles pour calculer cette complexité, nous obtenons :

- 1, 5, 8, 11 et 13, sont en $O(1)$
- le `if (10-11)` est en $O(1)$
- le `for (9-11)` imbriqué est en $O(n-i)$ et peut être borné par $O(n)$
- le `for global (6-13)` est en $O(n^2)$
- la procédure `tri_selection (1-13)` est en $O(n^2)$

9.3.4 Complexité du tri par insertion

En analysant l'algorithme de tri par insertion, on peut voir que sa complexité maximale est également en $O(n^2)$. On peut voir que la complexité minimale est rencontrée dans le cas où la liste est déjà triée et vaut $O(n)$

9.3.5 Complexité du tri Bulle

Ici aussi la complexité maximale est en $O(n^2)$. La complexité minimale est également en $O(n^2)$.

9.3.6 Complexité du tri par énumération

Si $m < n$, la complexité moyenne et maximale de ce tri est en $O(n)$ ce qui est remarquable.

9.3.7 Quelques exemples intéressants

Les exemples suivants viennent d'interrogation ou d'examens précédents où l'on considère que m, l, n sont des paramètres entiers positifs.

```
i = 1
while i < n**3 :
    i=i*2

for i in range(n):
    for j in range(m):
        for k in range(l):
            print('hello')

i = 2
while i < n:
    i=i*i
```

9.4 Complexité des méthodes de manipulation de séquences

Voir le site wiki python pour plus de détails:

<http://wiki.python.org/moin/TimeComplexity>

On suppose que n est le nombre d'éléments dans la liste et k la valeur (ex: longueur) du paramètre.

La complexité moyenne suppose que les paramètres sont générés uniformément de façon aléatoire.

9.4.1 Complexité des opérations sur les listes

De façon interne, une liste est représentée sous forme de vecteur où les composantes sont contiguës en mémoire et avec, en moyenne, de la place libre pour allonger la liste. La pire opération est donc une insertion ou suppression en début de liste car toutes les composantes doivent bouger. Malheureusement, au pire des cas, un simple `s.append(x)` peut demander de recopier toute la liste s'il n'y a plus de place libre juste après la liste `s`.

Opération	Complexité moyenne	Complexité maximale
Copy	$O(n)$	$O(n)$
Append	$O(1)$	$O(n)$
Insert	$O(n)$	$O(n)$
Get Item	$O(1)$	$O(1)$
Set Item	$O(1)$	$O(1)$
Delete Item	$O(n)$	$O(n)$
Iteration	$O(n)$	$O(n)$
Get Slice	$O(k)$	$O(k)$
Del Slice	$O(n)$	$O(n)$
Set Slice	$O(k+n)$	$O(n+k)$
Extend	$O(k)$	$O(n+k)$
Sort	$O(n \log n)$	$O(n \log n)$
Multiply	$O(nk)$	$O(nk)$
<code>x in s</code>	$O(n)$	$O(n)$
<code>min(s), max(s)</code>	$O(n)$	$O(n)$
<code>len(s)</code>	$O(1)$	$O(1)$

9.4.2 Complexité des opérations sur les ensembles

L'implémentation des ensembles est similaire à celle des dictionnaires.

Opération	Complexité moyenne	Complexité maximale
<code>x in s</code>	$O(1)$	$O(n)$
<code>s t</code>	$O(\text{len}(s) + \text{len}(t))$	$O(\text{len}(s) * \text{len}(t))$
<code>s & t</code>	$O(\max(\text{len}(s), \text{len}(t)))$	$O(\text{len}(s) * \text{len}(t))$
<code>s - t</code>	$O(\max(\text{len}(s), \text{len}(t)))$	$O(\text{len}(s) * \text{len}(t))$
<code>s ^ t</code>	$O(\max(\text{len}(s), \text{len}(t)))$	$O(\text{len}(s) * \text{len}(t))$

9.4.3 Complexité des opérations sur les dictionnaires

La complexité moyenne suppose que les collisions sont rares avec la fonction de hashage (ce qui est généralement le cas).

Le cas moyen suppose que les paramètres sont choisis uniformément aléatoirement parmi les clés possibles.

Opération	Complexité moyenne	Complexité maximale
Copy	$O(n)$	$O(n)$
Get Item	$O(1)$	$O(n)$
Set Item	$O(1)$	$O(n)$
Delete Item	$O(1)$	$O(n)$
Iteration	$O(n)$	$O(n)$

LOGIQUE, INVARIANT ET VÉRIFICATION D'ALGORITHME

10.1 Introduction

Jusqu'à présent nous avons expliqué nos algorithmes en décrivant, de façon informelle, les différents traitements effectués par ceux-ci. Nous nous sommes "*assurés*" de leur bon fonctionnement en les testant sur certains exemples et, en particulier, sur des cas limites. Pour vérifier complètement, en utilisant des tests, qu'un algorithme fonctionne correctement dans tous les cas, il faudrait, en théorie, le tester avec toutes les données possibles, ce qui, en pratique, est généralement impossible vu le nombre énorme de données possibles.

Notons que ceci est vrai si nous nous limitons aux algorithmes séquentiels (une seule action à la fois), déterministes (2 exécutions de l'algorithme avec les mêmes données, donnent lieu exactement au même traitement). Si ce n'est pas le cas, le problème de test ou de vérification d'un algorithme est beaucoup plus complexe.

Ce chapitre présente, sur des exemples simples, le problème de la vérification d'un algorithme. Le problème général de vérification complète d'algorithme est complexe et ne possède pas de méthode systématique automatisable. Il est pourtant essentiel que le concepteur d'algorithmes soit conscient des problèmes et qu'il ait des notions de base en matière de vérification d'algorithmes. En effet, il est évident que la conception et la vérification d'algorithmes vont de pair. En particulier, la notion d'invariant de boucle, présentée ci-dessous, est essentielle pour comprendre comment fonctionne un programme.

Dans la suite de ce chapitre, après avoir fait un bref rappel de logique et avoir expliqué comment formaliser en logique, l'*état* d'un programme à un endroit donné de son exécution, nous allons énoncer le problème à résoudre en parlant de la notion de preuve partielle et totale, et ensuite, présenter la notion d'invariant de boucle qui, associée à la notion de terminaison de boucle, permet de vérifier un algorithme.

10.2 Rappel de logique

10.2.1 Le calcul des propositions

Syntaxe

Soit un ensemble P de propositions p, q, r, \dots . Nous définissons l'ensemble des formules f possibles en logique des propositions par les définitions en Backus Naur Form (BNF) suivante:

$$\begin{aligned} f &::= p \quad (\forall p \in \mathcal{P}) \\ f &::= \neg f \\ f &::= f \text{ op } f \\ f &::= (f) \end{aligned}$$

avec

$$\text{op} ::= \vee \mid \wedge \mid \rightarrow \mid \leftarrow \mid \leftrightarrow$$

(le ou inclusif, le et, et les implications à droite à gauche ou dans les deux sens).

Note: La définition précédente utilise la notation Backus Naur Form (BNF). BNF est un formalisme qui permet de définir un langage; par exemple l'ensemble des syntaxes possibles pour une formule logique ou pour un programme *Python*. Les définitions donnent un système de réécriture où la notation $f ::= p$ signifie f peut être réécrit en p . Quand f peut être réécrit en p ou q on écrit $f ::= p, f ::= q$ ou de façon plus compacte $f ::= p \mid q$.

Par exemple;

$$p \wedge q$$

est une formule correcte puisque la définition BNF permet la séquence de réécriture suivante:

$$\begin{aligned} &f \\ &f \text{ op } f \\ &f \wedge f \\ &p \wedge f \\ &p \wedge q \end{aligned}$$

De même les formules suivantes peuvent être obtenues:

$$\begin{aligned} &p \wedge (q \vee r) \\ &(p \rightarrow q) \leftrightarrow \neg r \end{aligned}$$

Sémantique

Une formule peut avoir la valeur vraie ou fausse suivant la valeur des différentes propositions. Nous dirons que le domaine sémantique d'une formule est l'ensemble $\{\text{True}, \text{False}\}$ (Vrai ou Faux) (dénoté aussi $\{\text{T}, \text{F}\}$). Nous parlons d'*interprétation d'une formule*, c'est-à-dire son évaluation, en ayant donné une valeur T ou F à chaque proposition.

La sémantique de chaque opérateur logique est définie grâce à sa *table de vérité*.

Par exemple: pour l'interprétation v où $v(p) = \text{F}$ et $v(q) = \text{T}$

$$\begin{aligned}v(p \wedge q) &= F \\v(p \leftrightarrow q) &= F \\v(p \rightarrow q) &= T\end{aligned}$$

Pour éviter de parenthéser complètement les formules nous posons:

$$\neg < \vee, \wedge < \leftarrow, \rightarrow, \leftrightarrow$$

où < signifie est plus prioritaire que.

Une formule logique est *consistante* si elle est vraie pour au moins une interprétation, et est *valide* ou *tautologique* si elle est vraie pour toutes les interprétations.

Deux formules f_1 et f_2 sont (logiquement) équivalentes, dénoté $f_1 \Leftrightarrow f_2$ (où \Leftrightarrow est un méta-opérateur) si pour toute interprétation v , $v(f_1) = v(f_2)$ c'est-à-dire si $f_1 \leftrightarrow f_2$ est toujours vrai.

Par exemple:

$$p \vee q \Leftrightarrow q \vee p$$

\Leftrightarrow est moins prioritaire que les autres opérateurs logiques.

Notons qu'une équivalence logique reste valable si l'on substitue partout une proposition par n'importe quelle formule (règle de substitution uniforme).

La liste suivante donne quelques équivalences logiques représentatives:

$$\begin{aligned}\neg\neg f &\Leftrightarrow f \\f \wedge f &\Leftrightarrow f \Leftrightarrow f \vee f \\f_1 \wedge f_2 &\Leftrightarrow f_2 \wedge f_1 \\f_1 \vee f_2 &\Leftrightarrow f_2 \vee f_1 \\f_1 \leftrightarrow f_2 &\Leftrightarrow f_2 \leftrightarrow f_1 \\f_1 \wedge (f_2 \wedge f_3) &\Leftrightarrow (f_1 \wedge f_2) \wedge f_3 \\f_1 \vee (f_2 \vee f_3) &\Leftrightarrow (f_1 \vee f_2) \vee f_3 \\T \wedge f &\Leftrightarrow f \\T \vee f &\Leftrightarrow T \\F \wedge f &\Leftrightarrow F \\F \vee f &\Leftrightarrow f \\f_1 \rightarrow f_2 &\Leftrightarrow \neg f_1 \vee f_2 \\f_1 \rightarrow T &\Leftrightarrow T \\f_1 \rightarrow F &\Leftrightarrow \neg f_1 \\T \rightarrow f_2 &\Leftrightarrow f_2 \\F \rightarrow f_2 &\Leftrightarrow T \\\neg(p \vee q) &\Leftrightarrow (\neg p) \wedge (\neg q) \\\neg(p \wedge q) &\Leftrightarrow (\neg p) \vee (\neg q)\end{aligned}$$

Enfin, si $f_1 \rightarrow f_2$ est valide (toujours vraie) nous disons que f_1 implique logiquement f_2 ou que f_1 est une condition suffisante pour f_2 et que f_2 est une condition nécessaire pour f_1 (notation $f_1 \Rightarrow f_2$ où \Rightarrow est un méta-opérateur).

\Rightarrow a la même priorité que \Leftrightarrow .

10.2.2 La logique des prédicats

Pour pouvoir exprimer des contraintes sur des variables de programmes, il faut ajouter aux formules logiques, la possibilité d'utiliser des variables simples ou indicées (listes, tuples...), d'effectuer des manipulations d'expressions arithmétiques et relationnelles. De plus il faut ajouter les *quantificateurs*

$$\forall(\text{pour tout}), \exists(\text{il existe})$$

Le format général d'une formule resp. pour tout et il existe est:

$$\begin{aligned} \forall x f(x) \\ \exists x f(x) \end{aligned}$$

où $f(x)$ est une formule qui utilise la variable x , variable qui est *liée par le quantificateur* existentiel (il existe) ou universel (pour tout).

Par opposition, les variables simples ou indicées qui ne sont pas liées par un quantificateur sont nommées *variables libres*.

$$\begin{array}{ll} x < y + 1 & : x \text{ et } y \text{ sont libres} \\ x = 7 \leftrightarrow y = 3 & : x \text{ et } y \text{ sont libres} \\ \forall x (x \cdot y = 0 \rightarrow x = 0) & : x \text{ est lié par } \forall x \text{ et } y \text{ est libre} \\ \forall x (x \cdot y = 0 \rightarrow x = 0 \vee y = 0) & : x \text{ est lié par } \forall x \text{ et } y \text{ est libre} \end{array}$$

Le pour tout et il existe sont moins prioritaires que la négation mais sont plus prioritaires que le et, ou, les implications, et les méta-opérateurs \Rightarrow et \Leftarrow . Notons que l'avant dernière formule est consistante (peut être vraie) tandis que la dernière est valide (est vraie pour toute interprétation).

De même, la formule :

$$\forall x (0 \leq x < 0 \rightarrow x = 7)$$

est valide (toujours vraie) puisque $0 \leq x < 0$ est toujours fausse.

Cette logique est la logique des *prédicats* ou logique *du premier ordre*.

Les notions d'équivalence et d'implication logiques sont étendues. La liste suivante donne quelques équivalences logiques représentatives où $f(x)$, $f_1(x)$ et $f_2(x)$ sont des formules utilisant la variable libre x :

1. $\forall x f(x) \Leftrightarrow \neg \exists x \neg f(x)$
2. $\exists x f(x) \Leftrightarrow \neg \forall x \neg f(x)$
3. Si le domaine des valeurs possibles pour x n'est pas vide:
 $\forall x f(x) \Rightarrow \exists x f(x)$ (sinon ce n'est pas vrai)
4. $\exists x (f_1(x) \vee f_2(x)) \Leftrightarrow (\exists x f_1(x) \vee \exists x f_2(x))$
5. $\forall x (f_1(x) \wedge f_2(x)) \Leftrightarrow (\forall x f_1(x) \wedge \forall x f_2(x))$

10.3 Exprimer l'état du programme formellement

Pour simplifier nos propos, nous n'allons travailler dans nos exemples, qu'avec des valeurs simples (généralement entières ou caractères) et des séquences de valeurs simples.

Le formalisme logique va servir à formaliser l'état d'un programme au cours de son exécution. Dans un premier temps, pour simplifier les explications, nous allons considérer du code séquentiel qui ne fait appel à aucune fonction.

Supposons par exemple que le programme manipule une variable x , on va construire des propositions grâce à des opérateurs relationnels. Par exemple

$$x = 3$$

représente une proposition vraie si x vaut 3 et fausse sinon (ici on utilise le symbole $=$ pour désigner l'égalité).

Par extension, la sémantique de cette formule, sera l'ensemble des états où cette formule est vraie, et donc l'ensemble des états du programme où x vaut 3, quelque soit la valeur des autres variables.

Exemple:

En pratique nous allons décrire l'état d'un programme à un moment de son exécution. Cet état est constitué par la valeur de chacune de ses variables (ou plus précisément des objets référencés par ses variables) et du "point de contrôle", c'est-à-dire l'endroit (instruction) où se trouve l'exécution du programme.

Par exemple dans le code :

```
x = 3           #1
y = 4           #2
z = x * y       #3
```

on peut voir l'état du programme à trois moments:

1. au début, c'est-à-dire avant qu'aucune instruction n'ait été exécutée,
2. après l'exécution de l'instruction #1,
3. après #2, et
4. après #3.

Pour exprimer ces états 1. à 4. les formules logiques "parleront" de x , y et z .

Après #1 on a

$$x = 3$$

Après #2, on a :

$$x = 3 \wedge y = 4$$

Après #3, on a :

$$x = 3 \wedge y = 4 \wedge z = 12$$

qui formalise la postcondition la plus forte (c'est-à-dire qui a le plus de contraintes) que l'on peut imaginer avec ce code.

Notons que l'état du programme avant la première instruction est, si aucune variable n'est initialisée, qu'aucune contrainte sur la valeur des variables n'existe: dans ce cas, formalisée par la formule `True` dont la sémantique est l'ensemble de tous les états possibles.

De façon générale, la sémantique d'une formule f pour un programme donné, est l'ensemble des états où f est vraie. Donc quand on exprime qu'avec une précondition donnée Pre , le code donne la postcondition $Post$, cela signifie que si l'on prend n'importe quel état de Pre en terme de valeur de variables du programme satisfaisant Pre , à la fin du code on sera dans un état satisfaisant $Post$ c'est-à-dire parmi l'ensemble des états satisfaisant le résultat $Post$.

Malheureusement les programmes, même séquentiels sans appels de fonctions, ne sont généralement pas si simples. En effet, le flux de l'exécution du programme dépend évidemment des données lues qui vont influencer les tests des instructions conditionnelles (`if`) et répétitives (`while`, et `for`).

S'il suffit de suivre la séquence d'exécution et l'effet sur les variables pour un jeu de données précis, cela n'a aucun sens en général car ce que l'on veut pouvoir faire s'exprimer l'état du programme en particulier à la fin de son exécution **quelles que soit les données traitées** pour déterminer si la Postcondition (résultat escompté) est toujours satisfaite.

Autre exemple:

Supposons le code suivant qui suppose que x et y sont des variables entières initialisées:

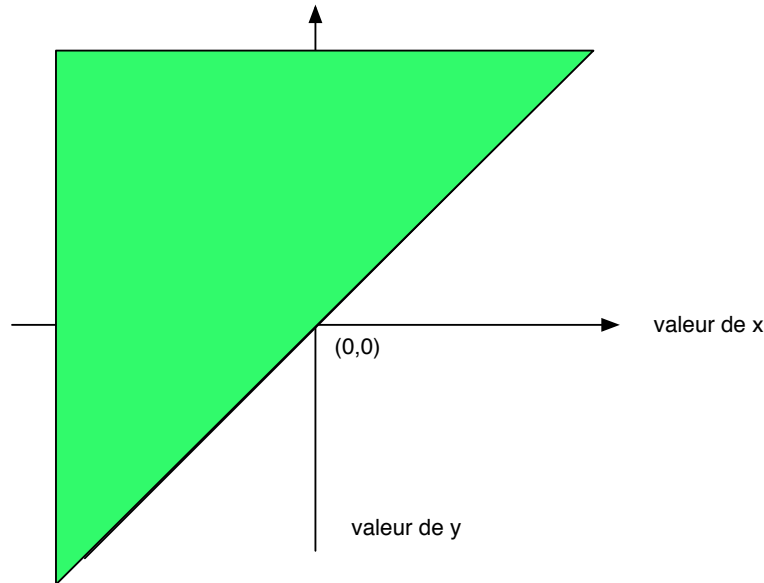
```
if x > y:
    x, y = y, x
```

On peut voir que la formule :

$$x \leq y$$

est une postcondition valide à ce code, c'est-à-dire que quelque soit la valeur initiale de x et y , on a à la fin de l'exécution le fait que la valeur de x est plus petit ou égal à la valeur de y .

En supposant x et y réels la figure suivante représente les valeurs possibles de x et de y :



Par ailleurs, pour comprendre comment s'exécute un programme, il est crucial de pouvoir décrire précisément son état au milieu de son exécution. Dans le cas d'une répétitive (while ou for), il est essentiel de pouvoir fournir une description unique de l'état du programme qui est valable au début de chaque itération ou plus précisément au moment où l'on teste la condition (du while ou du fait qu'il reste un élément à traiter par le for).

Ainsi dans la recherche séquentielle,

```
i = 0
while i < n and s[i] != x:
    i = i + 1
```

à chaque fois que le test du while est réalisé (que le résultat de ce test soit vrai ou faux), l'état du programme est décrit par la formule suivante:

$$(0 \leq i \leq n) \wedge (\forall j \ 0 \leq j < i \rightarrow s[j] \neq x)$$

qui exprime que i est entre 0 et n compris et que tous les éléments de $s[0:i]$ (i non compris) sont différents de x .

10.3.1 Exemples de formule exprimant une condition

Il est important de pouvoir exprimer en logique, ce que l'on désire comme résultat d'un code ou la postcondition la plus précise (c'est-à-dire la plus forte possible) d'un code donné. Il est également important de pouvoir, à partir de la spécification d'une postcondition donnée, comprendre ce qui est désiré et écrire le code correspondant.

Donnons des exemples de formalisation logique de conditions exprimées au départ en français:

Note: La plupart des exemples proviennent d'énoncés d'examen d'années précédentes.

x est dans l'intervalle entre 0 compris et n non compris

Formule correspondante :

$$0 \leq x < n$$

La valeur de x est inférieure ou égale à celle de tous les éléments d'une liste s (d'entiers):

Formule correspondante :

$$(\forall i \ 0 \leq i < \text{len}(s) \rightarrow x \leq s[i])$$

Notons ici, que x , s et $\text{len}(s)$ sont des variables (et fonctions) du programme *Python*. Par contre i est une variable liée au quantificateur logique pour tout. Ce n'est donc pas une variable du programme *Python*.

Au niveau de la formule logique, x , s (et $\text{len}(s)$) sont vues comme des variables *libres* tandis que i est une variable liée au quantificateur *pour tout* et ne correspond donc pas à une variable du programme.

Notez le canevas utilisé: on a un *pour tout* avec une variable dont on donne un nom de variable (si possible pas celui d'une variable du programme pour ne pas mettre de la confusion) et ensuite une implication logique :

$$\forall i \ \text{hypothese}(i) \rightarrow \text{conclusion}(i)$$

où $\text{hypothese}(i)$ et $\text{conclusion}(i)$ signifient que ce sont des formules qui utilisent la variable i .

L'hypothèse sert à identifier les endroits où il y a une contrainte. Par exemple ici, ce sont pour les valeurs de i entre 0 compris et $\text{len}(s)$ non compris.

La conclusion exprime la contrainte: ici x est inférieur ou égal à $s[i]$.

En conclusion la formule dit: pour tous les indices corrects i de s on doit avoir x plus petit ou égal à $s[i]$.

liste s triée par ordre non décroissant:

Formellement il suffit d'exprimer que les éléments voisins sont dans le bon ordre:

$$\forall i \ ((0 \leq i < \text{len}(s) - 1) \rightarrow (s[i] \leq s[i + 1]))$$

Notez que j'ai rajouté des parenthèses pour bien identifier les deux parties de l'implication, même si cela n'est pas nécessaire car l'implication est moins prioritaire que les comparaisons.

liste rangée de façon bizarre:

- Donnons une postcondition correcte d'un algorithme qui trie une liste avec les éléments d'indices pairs triés en ordre croissant et les éléments d'indices impairs, triés en ordre décroissant:

$$\begin{aligned} & \forall i ((0 \leq i < \text{len}(s-1)/2) \rightarrow (s[2i] \leq s[2i+2])) \\ & \quad \wedge \\ & \forall i ((0 \leq i < \text{len}(s)/2-1) \rightarrow (s[2i+1] \leq s[2i+3])) \end{aligned}$$

- Donnons une postcondition correcte d'un algorithme qui trie une liste avec les éléments pairs triés en ordre croissant et les éléments impairs, triés en ordre décroissant (attention, ce n'est pas comme le précédent !!) :

$$\begin{aligned} & \forall i, j ((0 \leq i < j < \text{len}(s) \wedge s[i]\%2 = 0 \wedge s[j]\%2 = 0) \rightarrow (s[i] \leq s[j])) \\ & \quad \wedge \\ & \forall i, j ((0 \leq i < j < \text{len}(s) \wedge s[i]\%2 = 1 \wedge s[j]\%2 = 1) \rightarrow (s[i] \geq s[j])) \end{aligned}$$

10.3.2 Exemple de formule exprimant une condition de code

Ayant un code, il peut être intéressant d'exprimer formellement ce qu'il fait (c'est-à-dire la postcondition) ainsi que la précondition nécessaire le plus précisément possible.

fonction logique

On peut par exemple demander en français et en logique d'exprimer la précondition et postcondition précise de ce code:

```
def test_liste(s):
    """
        précondition : ...
        postcondition: ...
    """
    delta = 0
    i = 0
    n = len(s)
    while i < n-1 and s[i] + delta <= s[i+1]:
        delta = s[i+1] - s[i]
        i = i+1
    return (i == n-1)
```

précondition : liste non vide. Formellement: $\text{len}(s) > 0$

Postcondition: renvoie vrai si s est trié par ordre croissant et qu'en le parcourant séquentiellement depuis le début, l'écart entre les valeurs successives reste constant ou augmente de plus en plus. Formellement:

$$\begin{aligned} & n = \text{len}(s) \\ & \quad \wedge \\ & \forall j ((0 \leq j < n-1) \rightarrow (s[j] \leq s[j+1])) \\ & \quad \wedge \\ & \forall j ((1 \leq j < n-1) \rightarrow (s[j] - s[j-1] \leq s[j+1] - s[j])) \end{aligned}$$

Encore un exemple

Exprimez en logique du premier ordre le résultat (valeur de *res*) de l'algorithme suivant qui travaille avec une liste *s* d'entiers:

```
n = len(s)
i = 1
res = 0
while i < n:
    if s[i] != s[res]:
        i = i+1
    else:
        res = res+1
        i = res+1
```

Postcondition:

$$\begin{aligned}
 & n = \text{len}(s) \\
 & \quad \wedge \\
 & 0 \leq \text{res} < n \\
 & \quad \wedge \\
 & \forall j ((0 \leq j < \text{res}) \rightarrow (\exists k (j < k < n \wedge s[j] = s[k]))) \\
 & \quad \wedge \\
 & \forall j ((\text{res} \leq j < n) \rightarrow (s[j] \neq s[\text{res}]))
 \end{aligned}$$

Notez que pour le quantificateur existentiel, le canevas général est :

$$\exists i \text{ hypothese}(i) \wedge \text{conclusion}(i)$$

où *hypothèse*(*i*) indique par exemple les conditions sur l'indice *i* et *conclusion*(*i*), la formule que l'on veut exprimer sur les données du programme.

10.3.3 Code satisfaisant une postcondition exprimée formellement

Généralement, la spécification formelle de ce que doit faire le code arrive avant celui-ci.

Par exemple on peut demander une fonction *f* qui reçoit un string *s* et dont la postcondition est:

$$\begin{aligned}
 & (\forall i (0 \leq i < \text{len}(s)) \rightarrow \exists j (0 \leq j < \text{len}(\text{res})) \wedge s[i] = \text{res}[j]) \\
 & \quad \wedge \\
 & (\forall i, j (0 \leq i < j < \text{len}(\text{res})) \rightarrow \text{res}[i] \neq \text{res}[j])
 \end{aligned}$$

Pouvez-vous me donner un tel algorithme en *Python* ?

10.4 Preuve partielle et totale

Etant donné qu'il est, en pratique généralement impossible de tester complètement le bon fonctionnement d'un algorithme, il faut se rabattre sur des méthodes analytiques pour le vérifier.

Essayons de prouver que le calcul du *pgcd* par la méthode d'Euclide, donné à la figure suivante donne le bon résultat en supposant qu'au départ $x \geq 0$ et $y > 0$.

```
def pgcd(x, y):
    """ calcule le pgcd(x,y) """
    while (y > 0):
        x, y = y, x % y
    return x
```

Deux problèmes doivent être résolus. Il faut s'assurer:

1. que pour tout couple de données (x,y) possible la fonction *pgcd* se termine,
2. que si cette fonction se termine, elle donne le bon résultat.

Nous dirons que pour faire la *preuve totale* d'un algorithme, il faut vérifier le point 2., c'est-à-dire en faire sa *preuve partielle* et prouver le point 1., c'est-à-dire vérifier la *terminaison*.

Pour un algorithme sans boucle ni appel de fonction, la preuve totale est relativement simple puisque l'on peut suivre, instruction par instruction, son évolution en découpant les différents cas possibles suivant les tests effectués tout au long de son exécution.

Le problème provient des boucles où le même code peut être utilisé plusieurs fois, le nombre de répétitions variant suivant les données.

La formule logique qui décrit la situation des boucles est appelée invariant. Intuitivement, l'invariant d'une *while* (ou d'un *for*) donne l'état du programme chaque fois que l'on effectue le test associé (que le résultat du test soit vrai ou faux). Ainsi la formule donnée dans l'exemple de la recherche séquentielle est un invariant. Formellement, pour que cette formule soit un invariant correct trois conditions doivent être remplies:

Warning: Pour effectuer la preuve partielle d'une boucle, il faut tout d'abord trouver un *invariant de boucle* c'est-à-dire une *affirmation* ou *assertion* vraie à chaque fois que l'exécution du programme atteint le début de chaque itération de la boucle et qui permet de déduire le résultat voulu, si l'exécution de la boucle *while* se termine.

Une formule *I* est un invariant si:

1. *I* est vrai au début de l'exécution de la boucle *while* au moment où l'exécution du programme atteint cet endroit.
2. Ayant *I* vrai et la condition *C* de continuation du *while* également vraie (et donc que le corps du *while* est évalué une fois de plus), il faut pouvoir vérifier qu'après l'exécution d'une itération du corps la boucle *while*, l'invariant *I* est à nouveau vrai.
3. Enfin pour effectuer la preuve partielle, cet invariant, composé avec le fait que la condition *C* de continuation du *while* devienne fausse, doit permettre de vérifier que la boucle *while* a effectivement "calculé" le résultat *R* qui lui était demandé.

Essayons de dégager une interprétation intuitive de la notion d'invariant. Si l'on regarde l'ensemble des situations dans lesquelles se trouve un algorithme au début de chaque itération d'une boucle, on remarque que les situations évoluent tout en ayant des points communs. Effectivement, la boucle est une façon de réaliser de façon itérative un certain traitement qui ne peut être réalisé en une fois. L'invariant sera généralement une façon d'exprimer que le *résultat escompté* (à la terminaison de la boucle) est égal à *ce qui a déjà été réalisé* lors des précédentes itérations de la boucle *plus ce qui reste encore à réaliser*. Au début de la boucle, ce qui est

réalisé est vide, on aura alors *Résultat* = *Ce qui reste à réaliser*. A la terminaison de la boucle, la condition d'arrêt de la boucle permet de vérifier que ce qui reste à réaliser est vide. On aura alors *Résultat* = *Ce qui est réalisé* et la preuve partielle pour la boucle sera réalisée.

Formellement pour un code `while C: instruction`, les trois conditions suivantes doivent être vraies :

$$\begin{aligned} &\text{code avant le while} \Rightarrow I \\ &I \wedge C + \text{corps du while exécute une itération} \Rightarrow I \\ &I \wedge \neg C \Rightarrow R \end{aligned}$$

Pour exprimer l'invariant de la boucle *while* de la fonction *pgcd*, il faut utiliser des notations permettant de distinguer les valeurs initiales de *x* et de *y* de leur valeur courante.

Nous dénoterons x_i , y_i les valeurs respectivement de *x* et *y* après la *i*ème itération, (initialement $x = x_0$, $y = y_0$) et *x*, *y* leur valeur courante.

L'invariant vaut alors:

$$\textbf{Invariant: } (pgcd(x_0, y_0) = pgcd(x, y)) \wedge (x \geq 0) \wedge (y \geq 0)$$

Cette formule exprime le fait que pour calculer $pgcd(x_0, y_0)$ il reste à calculer $pgcd(x, y)$ (ici on peut oublier ce qui a déjà été réalisé). Le progrès provient du fait qu'il est plus simple de calculer $pgcd(x, y)$ que $pgcd(x_0, y_0)$.

Vérifier un invariant se fait par récurrence. Pour le *pgcd*,

- **Base:** : Initialement, l'invariant est trivialement vrai puisque $x=x_0$ et $y=y_0$ avec x_0 et $y_0 \geq 0$.
- **Induction:** Si la formule est vraie au début de la *i*ème itération et que la boucle effectue une itération supplémentaire alors l'invariant est vrai au début de la *i+1*ème itération.

En effet, on peut montrer que

$$x > 0 \wedge y > 0 \Rightarrow pgcd(x, y) = pgcd(y, x \% y)$$

De plus

$$x > 0 \wedge y > 0 \Rightarrow x \% y \geq 0$$

Et donc, étant donné qu'après chaque itération, les nouvelles valeurs de *x* et *y* valent respectivement l'ancienne valeur de *y* et l'ancienne valeur de *x* modulo l'ancienne valeur de *y* (ce que l'on peut noter $x_i = y_{i-1}$, $y_i = x_{i-1} \% y_{i-1}$) on voit que l'invariant est vrai au début de la *i+1*ème itération.

Pour compléter la preuve partielle il faut montrer que l'invariant et la terminaison implique le résultat.

En effet:

$$(pgcd(x_0, y_0) = pgcd(x, y)) \wedge (x \geq 0) \wedge (y \geq 0) \wedge (y \leq 0) \Rightarrow pgcd(x_0, y_0) = x$$

Enfin, pour compléter la preuve totale il faut encore montrer que la boucle se termine effectivement.

Ayant $x_0 > 0$ et $y_0 \geq 0$ on peut voir qu'à chaque itération, la valeur (entière) y diminue d'au moins une unité. De ce fait le test ($y > 0$) sera assurément faux un moment donné.

Fonction de terminaison

En général, on vérifie qu'une boucle se termine toujours en mettant en évidence une *fonction de terminaison*, c'est-à-dire une fonction f à valeur entière qui dépend de la valeur des variables et telle que:

- la valeur de la fonction décroît strictement d'au moins une unité à chaque itération;
- si l'invariant de la boucle est vrai, f est positif ou nul.

L'invariant étant par définition vrai au début de chaque itération de la boucle, on est alors assuré de la terminaison de celle-ci. Comme pour la recherche d'un invariant correct, la recherche d'une telle fonction de terminaison ne possède pas de méthode systématique.

10.5 Exemples de preuves d'algorithmes

Cette section analyse brièvement différents algorithmes vus précédemment et donne les différents invariants des boucles.

10.5.1 minimum

Dans la fonction de recherche de l'indice du minimum, pour trouver l'invariant de la boucle `for`, il faut traduire ce `for` en un `while`. On obtient le code suivant:

```
res = 0
i = 1
while i < n:
    if V[i] < V[res]:
        res = i
    i += 1
```

et l'invariant du `while` permettant d'effectuer la preuve totale de l'algorithme est donnée ci-dessous:

$$I \equiv 0 < i \leq n \wedge \forall j (1 \leq j \leq i - 1 \rightarrow V[i_{min-provisoire}] \leq V[j])$$

Notons bien que I reste vrai même quand la condition du `while` `i < n` devient fausse. Une fonction de terminaison qui, associée à l'invariant, permet de faire la preuve totale, vaut: $n-i$

Note: Trouver l'invariant d'une boucle peut être difficile puisqu'aucun algorithme systématique n'existe pour le trouver. Dans le cadre de ce cours, nous vous demandons seulement de pouvoir prouver qu'un invariant que l'on vous donne est correct (c'est-à-dire remplit les 3 conditions données plus haut).

10.5.2 Autre exemple:

On donne le code suivant avec la fonction `incseq` où `v` est une liste d'éléments simples :

```
def incseq(v):
    n = len(v)
    i = 0
    k = 0
    maxseq = 0
    while i < n-1:
        seq = 1
        j = i
        while j < n-1 and v[j] <= v[j+1]:
            seq += 1
            j += 1
        if seq > maxseq:
            maxseq = seq
            k = i
        i += seq
    return v[k:k+maxseq]
```

Que fait cette fonction ? Montrez que la formule suivante :

$$\begin{aligned}
 &0 \leq k \leq k + \text{maxseq} \leq i \leq n \\
 &\quad \wedge \\
 &\forall m. k \leq m < k + \text{maxseq} - 1 \Rightarrow v[m] \leq v[m+1] \\
 &\quad \wedge \\
 &\forall p, q. ((0 \leq p \leq q < i) \wedge (\forall r. p \leq r \leq q-1 \Rightarrow v[r] \leq v[r+1])) \Rightarrow q - p < \text{maxseq}
 \end{aligned}$$

est un invariant correct pour la boucle `while` principale.

10.5.3 Tri par sélection

Le code suivant donne le tri par sélection. Les commentaires donnent les invariants des boucles.

```
def tri_selection(s):
    """
    Trie la liste s par sélection
    """
    n = len(s)
    i = 0
    while i < n-1: # invariant I2 (voir plus bas)
                    # fonction de terminaison 2: n-1-i
        # recherche du min dans V[i..n]
        min = i # min = indice du min provisoire
        j = i+1
        while j < n: # invariant I1 (voir plus bas)
                    # fonction de terminaison 1: n-j
            if V[j][0] < V[min][0]:
                min = j
            j = j+1
        # placement du ième élément: échange V[i] <-> V[min]
```

```
V[min],V[i] = V[i],V[min]
i = i+1
```

avec

- invariant $I2$:

$$0 \leq i < n \wedge \\ \forall k (0 \leq k < i - 1 \rightarrow V[k] \leq V[k + 1]) \wedge \\ \forall m \forall k (0 \leq m \leq i - 1 \wedge i \leq k < n \rightarrow V[m] \leq V[k])$$

- invariant $I1$:

$$(i < j \leq n) \wedge \forall k (i \leq k < j \rightarrow V[min] \leq V[k])$$

10.5.4 Tri par insertion

Le code suivant donne le tri par insertion.

```
def tri_insertion(s):
    """
    Trie liste s par insertion
    """
    n = len(s)
    i = 1
    while i < n: # invariant I2 (voir plus bas)
        # fonction de terminaison 2: n-i
        Save = V[i] # utilisé pour l'échange
        # insertion de V[i]
        j = i-1
        while j >= 0 and V[j][0] > Save[0]: # invariant I1 (voir plus bas)
            # fonction de terminaison 1: j
            V[j+1] = V[j]
            j=j-1
        V[j+1] = Save
        i = i+1
```

avec

- invariant $I2$:

$$0 < i \leq n \wedge \\ \forall k (0 \leq k < i - 1 \rightarrow V[k] \leq V[k + 1])$$

- invariant $I1$:

$$-1 \leq j \leq i - 1 \wedge \\ \forall k (j + 2 \leq k \leq i \rightarrow V[k] > Save)$$

RÉCURSIVITÉ

See Also:

Référence: livre de Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein - Algorithmique

Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein - Algorithmique - 3ème édition - Cours avec 957 exercices et 158 problèmes, Dunod, 2010, ISBN: 978-2-10-054526-1 <http://www.books-by-isbn.com/2-10/2100545264-Algorithmique-Cours-avec-957-exercices-et-158-problemes-2-10-054526-4.html>

11.1 Motivation et introduction du concept

Un algorithme est dit récursif s'il s'appelle lui-même directement ou indirectement via l'appel d'une ou de plusieurs autres fonctions qui elles-mêmes finissent par l'appeler.

La récursivité est un concept fondamental en informatique qui met naturellement en pratique un mode de pensée puissant qui consiste à pouvoir découper la tâche à réaliser en sous-tâches de mêmes natures mais plus petites qui finalement sont simples à résoudre.

La démarche s'inspire de la démarche inductive mathématique utilisée dans les preuves par récurrence:

Rappel du principe

Si l'on doit prouver l'assertion $S(n) = f(n)$ pour tout n supérieur ou égal à n_0

Pour le cas de base on prouve l'assertion pour certaines petites valeurs de n (n_0 par exemple)

Pour l'étape de récurrence : on suppose que c'est vrai pour n inférieure ou égale à k (avec k supérieure ou égale à n_0), et prouve que c'est également vrai pour $n = k + 1$

Remarque : on peut aussi supposer que c'est vrai pour n inférieur ou égal à $k - 1$ et prouver que c'est vrai pour $n = k$.

C'est ce qu'on fera ici : souvent plus intuitif d'un point de vue informatique.

Prenons par exemple le calcul de la factorielle d'un nombre n . Par définition pour un n entier strictement positif, $n!$ est égale au produit des entiers strictement positifs inférieurs à n . Par convention on a aussi $0! = 1$.

Rappelons le code itératif d'une fonction calculant la factorielle:

```
def fact(n):
    res = 1
    i = 1
    for i in range(1, n+1):
        res = res * i
    return res
```

La définition récursive se base sur la définition $n! = n.(n-1)!$ pour tout $n > 0$ avec $0! = 1$

On obtient le code:

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

ou plus court:

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

Notons que nous avons déjà vu plusieurs algorithmes écrits de façon itérative mais dont l'idée de base était récursive.

Le programme de test de la conjecture de syracuse:

```
def test_syracuse(n):
    while n != 1:
        if n % 2 == 0:
            n = n//2
        else:
            n = n*3+1
    return True
```

s'énonce à chaque étape si n est pair on le divise par 2 sinon on le multiplie par 3 et ajoute 1.

Littéralement on pourrait l'écrire:

```
def test_syracuse(n):
    if n == 1:
        res = True
    else:
        if n%2 == 0:
            n = n//2
        else:
            n = 3*n+1
        res = test_syracuse(n)
    return res
```

ou de façon plus abrégée:

```
def test_syracuse(n):  
    if n == 1:  
        res = True  
    else:  
        res = test_syracuse(n//2 if n%2 == 0 else 3*n+1)  
    return res
```

ou encore plus abrégé (ce qui devient illisible):

```
def test_syracuse(n):  
    return True if n == 1 else test_syracuse(n//2 if n%2 == 0 else 3*n+1)
```

Note: En pratique, le nombre d'appels imbriqués de fonction est limité, par défaut, à 1000 appels. Par exemple, l'exécution de la version récursive de `fact(1000)` provoque l'erreur `RuntimeError: maximum recursion depth exceeded in comparison`.

Le code pour la *recherche dichotomique* sur une liste de couples triés sur la clé (premier élément des couples):

```
def recherche_dichotomique(s, x):  
    bi, bs = 0, len(s)  
    m = (bi+bs)//2  
    while bi < bs and x != s[m][0]:  
        m = (bi+bs)//2  
        if s[m][0] < x:  
            bi = m+1  
        else:  
            bs = m # x est avant ou est trouvé  
  
    if len(s) <= m or s[m][0] != x: # pas trouvé  
        m = -1  
    return m
```

peut s'écrire récursivement

```
def recherche_dichotomique(s, x, bi, bs):  
    if bi >= bs:  
        res = -1  
    else:  
        m = (bi+bs)//2  
        if s[m][0] < x:  
            res = recherche_dichotomique(s, x, m+1, bs)  
        elif s[m][0] > x:  
            res = recherche_dichotomique(s, x, bi, m)  
        else:  
            res = m  
    return res
```

11.2 Mécanismes

Prenons un simple code récursif (fonction *foo*) dont le cas de base est celui où le paramètre *n* vaut 0 et qui sinon fait un appel récursif à une instance *foo(n-1)*.

```
def foo(n):  
  
    if n == 0:  
        # traitement cas de base  
    else  
        ...  
        foo(n-1)  
        ...
```

La fonction *foo* réalise généralement un certain traitement pour le cas de base, et peut effectuer un traitement avant (pré-traitement), ou après (post-traitement) l'appel récursif.

Les trois codes suivants illustrent ces possibilités où les traitements correspondent simplement à des `print` :

Avec pré-traitement:

```
def foo(n):  
    if n==0:  
        print("cas de base :", n)  
    else:  
        print("pre-traitement pour : " , n)  
        foo(n-1)
```

`foo(5)`

donne:

```
pre-traitement pour : 5  
pre-traitement pour : 4  
pre-traitement pour : 3  
pre-traitement pour : 2  
pre-traitement pour : 1  
cas de base : 0
```

Avec post-traitement:

```
def foo(n):  
    if n==0:  
        print("cas de base")  
    else:  
        foo(n-1)  
        print("post-traitement pour : " , n)
```

`foo(5)`

donne:

```
cas de base : 0
post-traitement pour : 1
post-traitement pour : 2
post-traitement pour : 3
post-traitement pour : 4
post-traitement pour : 5
```

Avec pré et post-traitement :

```
def foo(n):
    if n==0:
        print("cas de base : ", n)
    else:
        print("pre-traitement pour : " , n)
        foo(n-1)
        print("post-traitement pour : " , n)
```

```
foo(5)
```

donne:

```
pre-traitement pour : 5
pre-traitement pour : 4
pre-traitement pour : 3
pre-traitement pour : 2
pre-traitement pour : 1
cas de base : 0
post-traitement pour : 1
post-traitement pour : 2
post-traitement pour : 3
post-traitement pour : 4
post-traitement pour : 5
```

Notons qu'ici on suppose que l'instance n appelle une fois l'instance $n-1$. Il se peut qu'elle appelle plusieurs instances $n-1$ comme nous le verrons dans certains exemples plus bas.

Note: Quand une instance donnée (par exemple de paramètre n) de la fonction *foo* appelle récursivement une instance plus petite (par exemple de paramètre $n-1$, on dit généralement que l'instance *père* *foo*(n) appelle une instance *fil* *foo*($n-1$).

Illustrons ces trois canevas.

11.2.1 Fonction récursive avec pré-traitement

Le code récursif du test de *Syracuse* ainsi que celui de la *recherche dichotomique* sont des exemples de code avec uniquement un pré-traitement ou un traitement de base (excepté le return du résultat obtenu qui constitue un post-traitement très simple). Dans les deux codes, on fait un test en début de fonction et ensuite on appelle récursivement la fonction avec des paramètres modifiés.

Les deux exemples suivant illustrent aussi bien un traitement récursif avec pré-traitement:

Calcul de toutes les permutations des éléments d'une séquence

L'idée est simple: l'ensemble des permutations des éléments d'une séquence *s* est donné en prenant l'ensemble des séquences commençant par un symbole quelconque de *s* suivi de toutes les permutations des symboles restants.

On obtient facilement le code suivant:

```
def Permutations(prefixe, s):
    if len(s) == 0: # prefixe est une nouvelle permutation
        print(prefixe)
    else: # prend un élément de la séquence comme suivant
          # et construit la suite avec ce qu'il reste à permuter
        for i in range(len(s)):
            Permutations(prefixe + s[i], s[:i]+s[i+1:])

Permutations("", "abcd")
print()
Permutations("", ("belle Marquise ", "vos beaux yeux ", "me font ", \
                 "mourir ", "d'amour ", "pour vous "))
```

On peut aussi stocker les résultats dans une liste:

```
def Permutations(prefixe, s, l):
    if len(s) == 0:
        l.append(prefixe)
    else:
        for i in range(len(s)):
            Permutations(prefixe + s[i], s[:i]+s[i+1:], l)

# fact est utilisé pour l'impression du résultat
def fact(n):
    return 1 if n==0 or n == 1 else n * fact(n-1)

sequence= input("séquence : ") # lecture de la séquence
                                # de caractères dont on veut les permutations
liste = [] # contiendra les résultats construits par la fonction Permutations
Permutations("", sequence, liste)

# imprime les résultats
for i,j in zip(range(fact(len(sequence))),liste):
    print("{0:5d} : {1}".format(i,j))
```

Tri rapide (Quicksort)

Le tri rapide est également un algorithme récursif avec pré-traitement

L'idée est que pour trier un vecteur (liste python), on choisit au hasard un (par exemple le premier) élément du vecteur : il sera appelé l'élément *pivot*. Ensuite on classe les éléments du vecteur de manière à placer avant le pivot les éléments qui lui sont inférieurs, et après ceux qui lui sont supérieurs. On appelle (récursivement) le tri rapide sur les deux moitiés de part et d'autre du pivot. Lorsque la séquence à trier est de taille 0 ou 1, il n'y a plus rien à faire, car le tableau est déjà trié.

Le tri rapide sera donné ultérieurement dans un cours d'algorithmique.

Une implémentation naïve du quicksort donne:

```
def quick_sort(s):
    """ Tri rapide (quicksort) de s (liste de couples (clé, valeur)) """
    if len(s) <= 1:
        res = s
    else:
        pivot = s[0]
        prefixe = quick_sort([x for x in s[1:] if x[0] < pivot[0]])
        suffixe = quick_sort([x for x in s[1:] if x[0] >= pivot[0]])
        res = prefixe + [pivot] + suffixe
    return res
```

Notez qu'ici *s* n'est pas modifié, mais que la fonction renvoie une nouvelle liste triée. Une version *en place* et plus efficace du Tri rapide est possible.

11.2.2 Fonction récursive avec post-traitement:

Dans ce cas, souvent quand pour calculer le résultat de l'instance père de la fonction, on doit d'abord avoir la valeur de son ou ses fils.

La fonction récursive *fact(n)* en est un exemple très simple.

Donnons en deux autres exemples.

Evaluation d'expressions arithmétiques

C'est un exemple typique de post-traitement: par exemple, $3 + 4 * 5$ prend la valeur 3 (dont on a directement la valeur) et $(4 * 5)$ évaluée avant, pour finalement avoir le résultat de la somme.

```
def evaluate(v):
    if type(v) is not list:
        res = v
    elif v[1] == '+':
        res = evaluate(v[0]) + evaluate(v[2])
    elif v[1] == '-':
        res = evaluate(v[0]) - evaluate(v[2])
    elif v[1] == '*':
        res = evaluate(v[0]) * evaluate(v[2])
    elif v[1] == '/':
        res = evaluate(v[0]) / evaluate(v[2])
```

```
elif v[1] == '^':
    res = evaluate(v[0]) ** evaluate(v[2])
else:
    res = None # error
return res
```

Tri fusion (Mergesort)

Un autre exemple de récursivité avec post-traitement est le Tri fusion. On divise la séquence de n éléments à trier en deux sous-séquences de $n/2$ éléments, on trie les deux sous-séquences à l'aide du tri fusion (appel récursif), on combine, en les fusionnant, les deux sous-séquences triées pour produire la séquence triée.

Le cas de base est le tri d'une séquence de taille 1.

Le code suivant est une version non optimisée du Tri fusion:

```
def merge(t1, t2):
    if len(t1) == 0:
        res = t2
    elif len(t2) == 0:
        res = t1
    elif t1[0][0] < t2[0][0]:
        res = [t1[0]] + merge(t1[1:], t2)
    else:
        res = [t2[0]] + merge(t1, t2[1:])
    return res

def merge_sort(t):
    if len(t) > 1:
        (t1, t2) = (t[:len(t)//2], t[len(t)//2:])
        t1 = merge_sort(t1)
        t2 = merge_sort(t2)
        res = merge(t1, t2)
    else:
        res = t
    return res

l=list(zip("bonjour", range(10)))
res = merge_sort(l)
print(res)
```

Notez aussi que ici `l` n'est pas modifié, mais que la fonction renvoie une nouvelle liste triée.

11.3 Structures de données récursives

Le récursivité est intensivement utilisée dans des structures de données telles que les graphes ou les arbres informatiques.

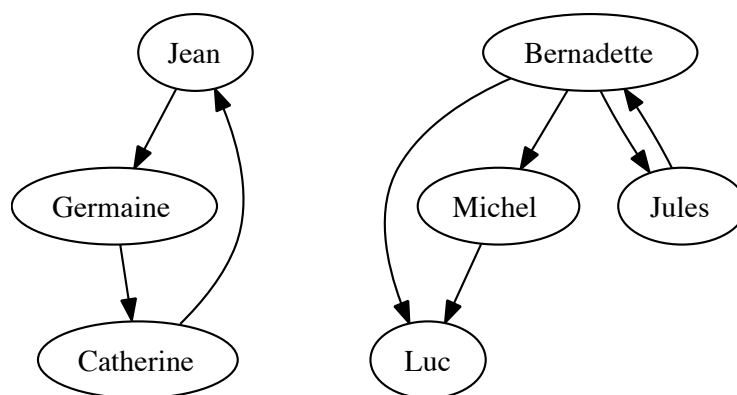
11.3.1 Graphe

Un graphe $G = (V, E)$ est une structure de données, formée d'un ensemble V d'éléments et d'un ensemble de paires (ou de couples) établissant les liens entre ces éléments.

Si E est un ensemble de paires, on dit que le graphe est *non orienté*, s'il s'agit de couples, le graphe est orienté.

Le graphique suivant décrit un graphe orienté donnant un ensemble V de personnes ($V = \{Jean, Germaine, Catherine, Luc, Bernadette, Jules, Michel\}$) avec la relation E (connaît).

$E = \{ Jean \rightarrow Germaine ; Germaine \rightarrow Catherine ; Catherine \rightarrow Jean ; Bernadette \rightarrow Luc ; Bernadette \rightarrow Michel ; Michel \rightarrow Luc ; Bernadette \rightarrow Jules ; Jules \rightarrow Bernadette \}$



Une façon simple d'implémenter un tel graphe en python est avec un dictionnaire dont les éléments sont les clés, et la liste de leurs connaissances, leur valeur. Dans l'exemple plus haut on aurait:

```

graphe = { "Jean"      : ["Germaine"] ,
           "Germaine" : ["Catherine"] ,
           "Catherine": ["Jean"] ,
           "Luc"      : [],
           "Michel"   : ["Luc"],
           "Bernadette": ["Luc", "Michel", "Jules"],
           "Jules"    : ["Bernadette"] }

```

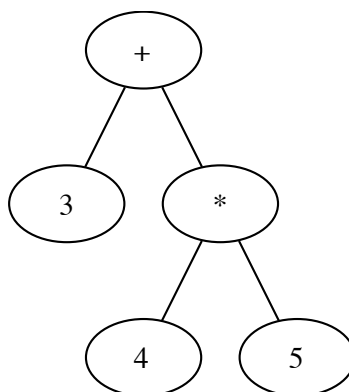
11.3.2 Arbre binaire

L'exemple de l'expression à évaluer, vu plus haut, est en fait une implémentation *python*, avec des listes, de la notion d'arbre binaire.

Un **arbre binaire** est une structure de données soit vide soit formée d'un *noeud* et de deux sous-arbres binaires, appelés respectivement *fil gauche* et un *fil droit*.

Chaque noeud contient une information appelée *contenu*.

Ainsi l'expression $3+4*5$ est vu comme l'arbre binaire illustré par la figure suivante :



où les noeuds vides ne sont pas représentés.

Notre implémentation *python* donnait: `exp = [3, '+', [4, '*', 5]]`

Note: La récursivité ainsi que de nombreux algorithmes sur les arbres et les graphes seront vus plus en détails dans des cours d'algorithmiques ultérieurs.

11.4 Traduction de fonction récursive en fonction itérative

Toute récursivité peut plus ou moins facilement être *traduite* en fonction non récursive.

Dans le cas de récursivité terminale (tail recursion) c'est-à-dire où la dernière action de la fonction consiste en un appel récursif, il est assez simple de remplacer la récursivité par une boucle while.

Dans le cas où le traitement est uniquement un post-traitement, il est possible de prendre le code à "l'envers" c'est-à-dire, de traduire le code en boucle qui fait le traitement de base, suivi du post-traitement pour n valant 1 ensuite pour n valant 2, ... jusqu'à avoir traité le cas pour la valeur n initiale. C'est ce qui se passe pour le code non récursif de la factorielle.

Quand la fonction à plusieurs appels à des instances plus petites ou qu'il y a à la fois un pré-traitement et un post-traitement, il est plus difficile de supprimer la récursivité et ne peut souvent être fait qu'avec une pile (stack) simulant cette récursivité.

11.5 Gestion de la mémoire à l'exécution

11.5.1 Noms et objets

Nous avons déjà vu qu'une variable est une référence à un objet. Manipuler cet objet se fait grâce à cette variable qui en quelque sorte l'identifie.

Ainsi comme nous l'avons déjà expliqué :

```
a = 3
a = a + 1
```

crée un objet de classe entière (`int`) dont la valeur est 3, et place dans la variable `a` son adresse (sa référence). Ensuite l'addition crée un autre objet de type (classe) `int` et de valeur 4 et `a` reçoit la référence à ce nouvel objet.

```
>>> dir(a)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
 '__delattr__', '__divmod__', '__doc__', '__eq__', '__float__',
 '__floor__', '__floordiv__', '__format__', '__ge__',
 '__getattr__', '__getnewargs__', '__gt__', '__hash__',
 '__index__', '__init__', '__int__', '__invert__', '__le__',
 '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__',
 '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__',
 '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__',
 '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
 '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
 '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
 '__sub__', '__subclasshook__', '__truediv__', '__trunc__',
 '__xor__', 'bit_length', 'conjugate', 'denominator', 'from_bytes',
 'imag', 'numerator', 'real', 'to_bytes']
```

`dir(a)` renvoie la liste des attributs et méthodes de l'objet référencé par la variable `a`.

En fait en *Python* tout est objet (ou référence à un objet).

Dans le vocabulaire *Python*, une variable est appelé **nom** et est *liée* à un objet. Un objet peut avoir plusieurs noms, par exemple dans le code ci-dessous où l'objet de type liste est connu sous les noms `s` et `t` dans un code appelant et `x` dans la fonction `foo` appelée.

```
def foo(x):
    x[0] = 3

s = [0, 0, 7]
t = s
foo(s)
```

Tout objet *Python* `x` a un identificateur (donné par `id(x)`), un type (donné par `type(x)`), et un contenu.

Lors de son exécution, un programme *Python* gère des **espaces de noms** (**namespace**). Un espace de nom est un *mappage* (*mapping*) de noms vers des objets. Des exemples de tels

espaces de noms sont donnés par l'ensemble des noms globaux (donné par `globals()`) ou des noms locaux à une instance de fonction (donné par `locals()`).

Note:

- Les espaces de nom peuvent être implémentés sous forme de dictionnaires *Python*.
- Ainsi, l'assignation ainsi que le passage de paramètres à une fonction modifient les espaces de noms et pas les objets eux mêmes !
- Par contre :

```
x = []  
x.append(3)
```

crée une liste et un nom dans l'espace de nom courant et ensuite appelle la méthode `append()` qui va modifier l'objet *référéncé* par `x`.

11.5.2 Scope et espace de nom (namespace)

En pratique la gestion des objets et des espaces de nom en mémoire implique deux espaces mémoire:

1- l'un, appelé **pile d'exécution (runtime stack)** qui va contenir l'espace de nom global et les espaces de noms locaux;

2- l'autre, **tas d'exécution (runtime heap)**, qui contient les objets.

La gestion des espaces de nom locaux (ainsi que d'autres éléments en mémoire, nécessaires pour le bon fonctionnement du programme) est effectuée dans des *trames (frames)* dans la *pile d'exécution (runtime stack)*: chaque fois qu'une instance de fonction `foo()` est appelée, une *trame* associée à cette instance est créée et mise comme un élément supplémentaire de la pile d'exécution. Cette trame contient en particulier, l'espace de nom local à cette instance. Cette trame sur la pile d'exécution continuera à exister jusqu'à la fin de l'exécution de cette instance de `foo()`. Ensuite (au moment du `return`), la trame est enlevée de la pile système et on revient à l'instance appelante dont la trame se trouve maintenant au sommet de la pile d'exécution.

Par contre un objet créé, continue à exister jusqu'à ce qu'il ne soit plus relié à aucun nom dans un espace de noms quelconque et que le système décide de récupérer l'espace mémoire qui lui était attribué. Cette opération est nommée *garbage collection* et s'effectue de façon *transparente* pour le programmeur, si ce n'est qu'il peut observer un ralentissement ponctuel du programme pendant que le *garbage collector* s'exécute. C'est en raison de la gestion sans ordre de cet espace mémoire où se trouvent les objets *Python*, qu'il est appelé le *tas d'exécution (runtime heap)*.

Exemple d'exécution d'un programme

Prenons l'exécution du programme suivant qui est une version simplifiée du tri par fusion pour des listes de caractères, exécutée sur la liste `list("CDAB")`.

```

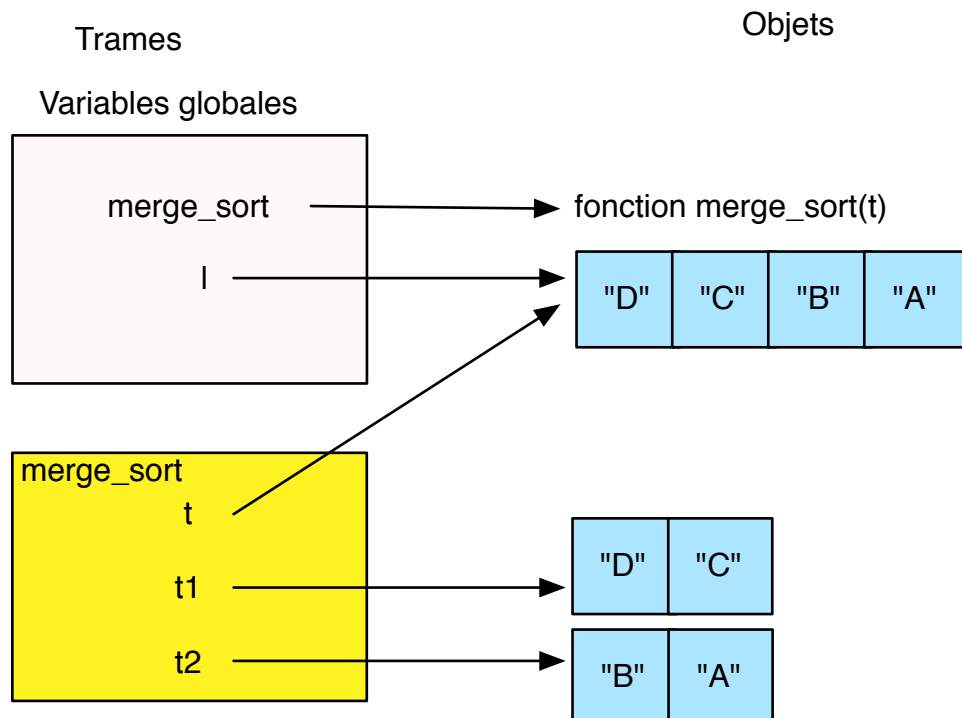
def merge_sort(t):
    if len(t) > 1:
        (t1, t2) = (t[:len(t)//2], t[len(t)//2:])
        t1 = merge_sort(t1)
        t2 = merge_sort(t2)
        # merge
        res = []
        while len(t1) > 0 and len(t2) > 0:
            if t1[0] < t2[0]:
                res.append(t1[0])
                del t1[0]
            else:
                res.append(t2[0])
                del t2[0]
        res += t1 + t2
    else:
        res = t
    return res

l = list("DCBA")
print(merge_sort(l))

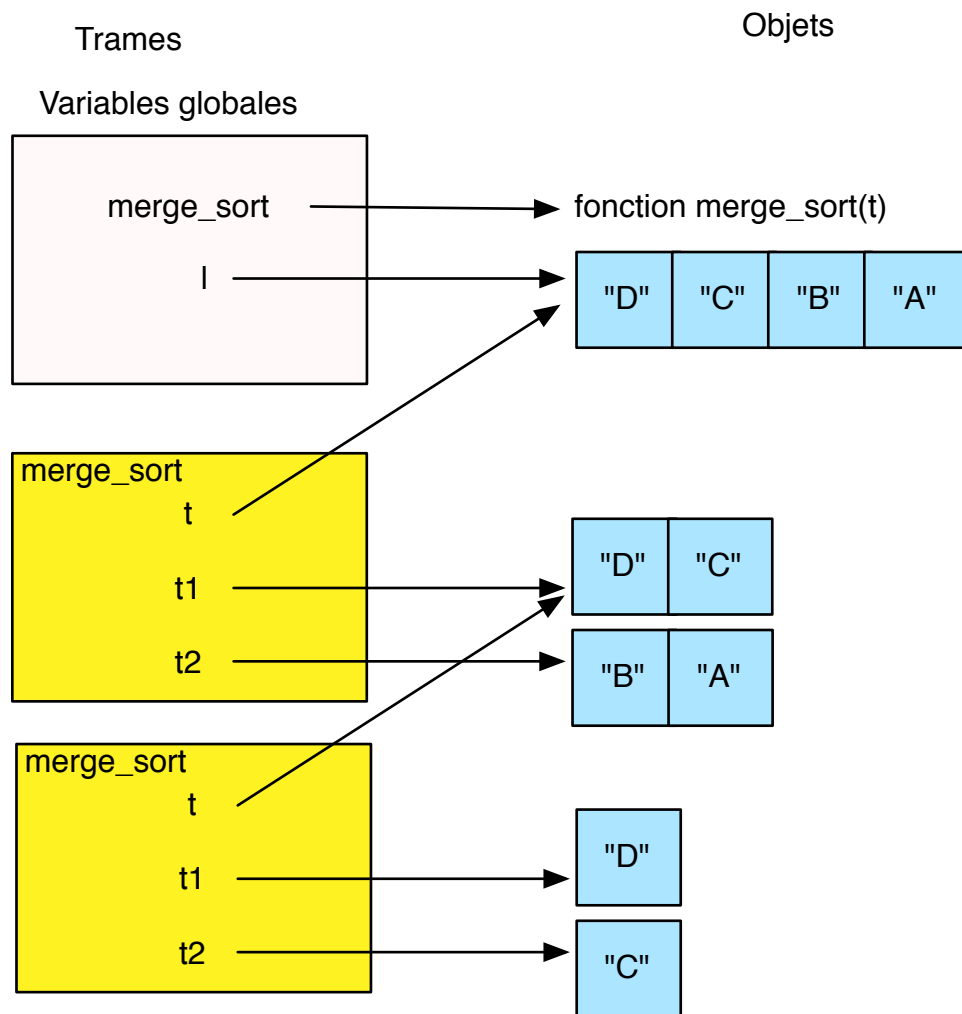
```

Illustrons en particulier les valeurs des variables `t1` et `t2` avant et après les appels récursifs et au moment des `return` (pour simplifier les diagrammes d'états, on représente par le caractère "X" la référence à ce caractère "X").

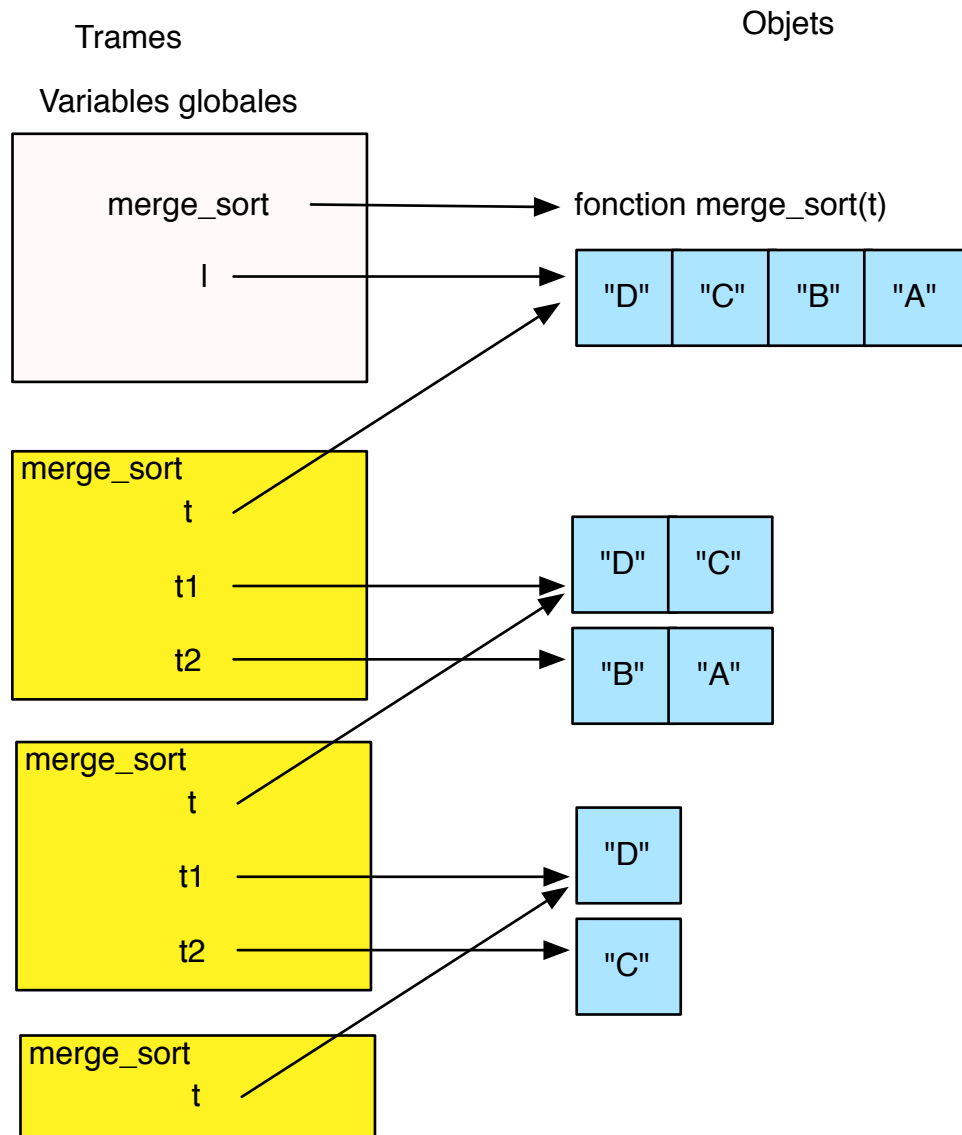
Etat dans l'instance de fonction de niveau 1 ("DCBA") avant les appels au niveau 2.



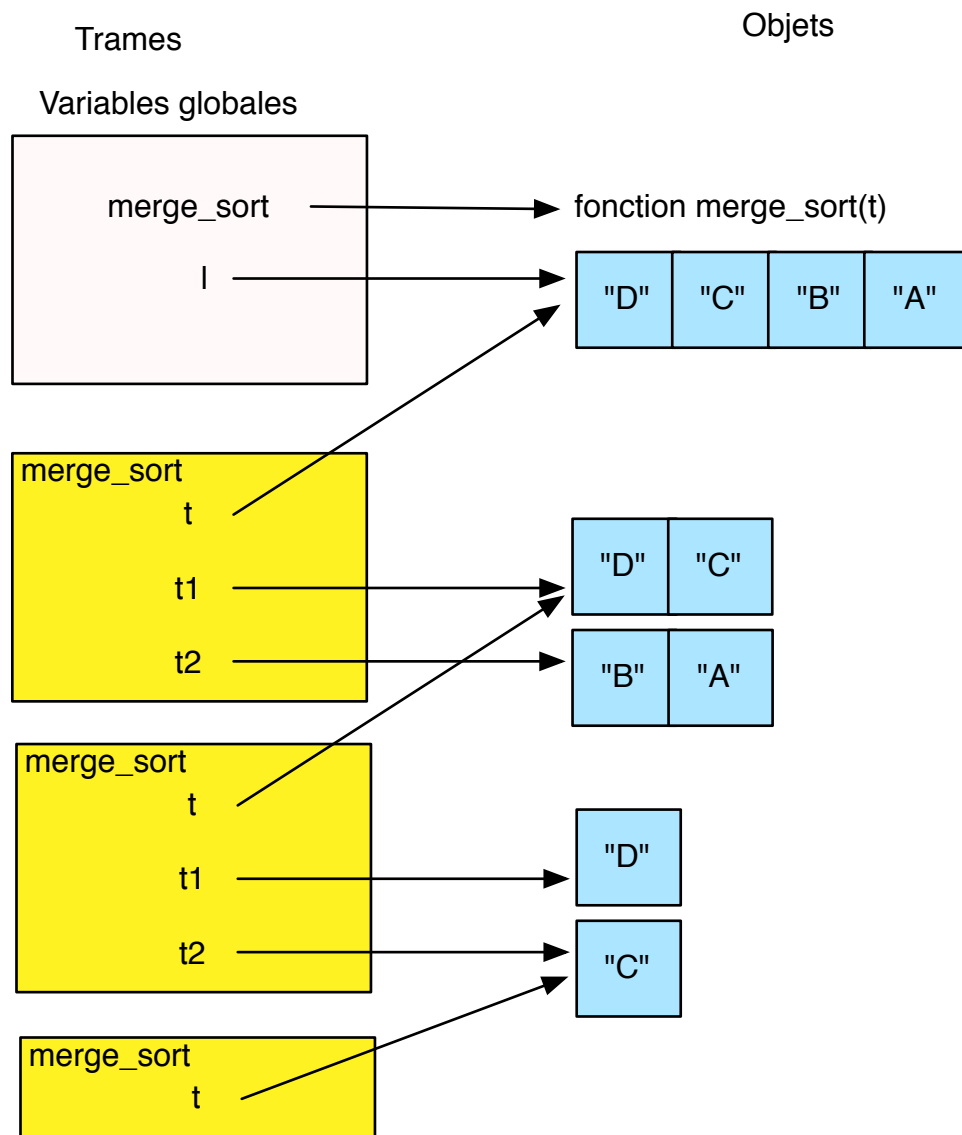
Etat dans l'instance de fonction de niveau 2 ("DC") avant les appels au niveau 3.



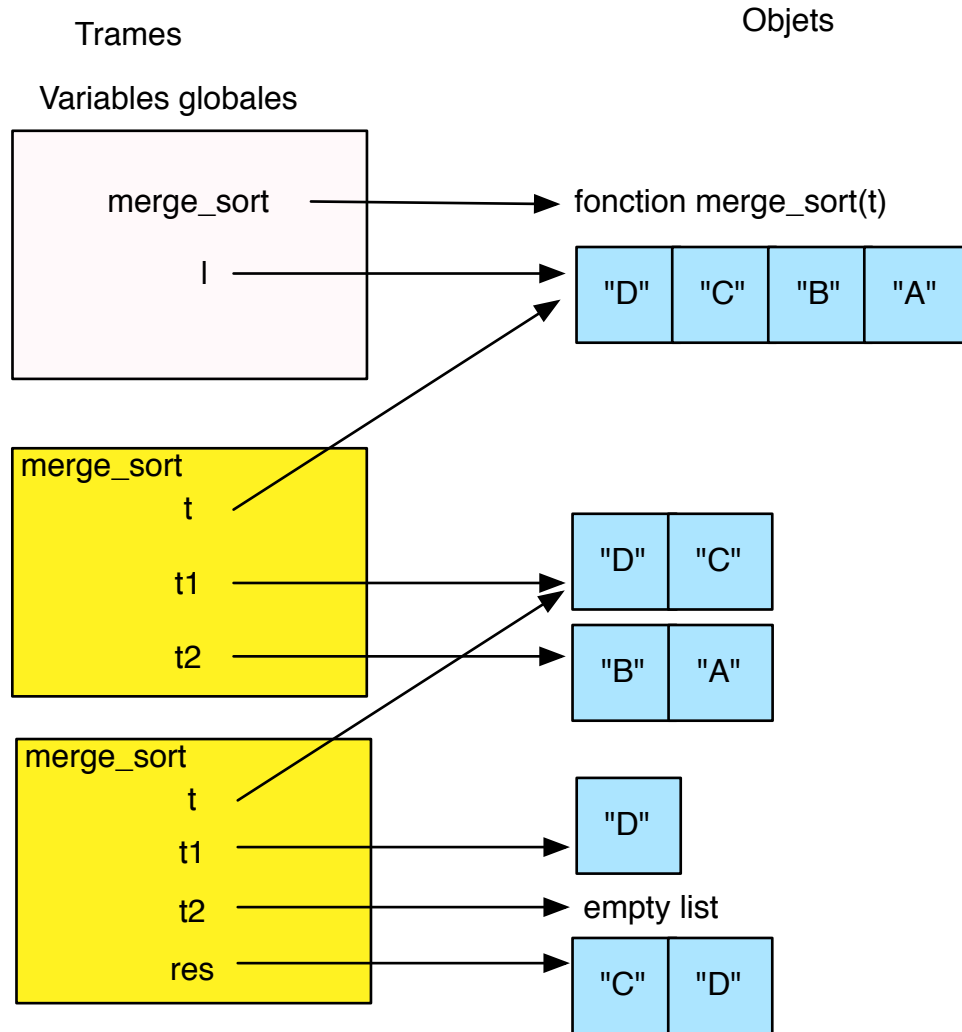
Etat dans l'instance de fonction de niveau 3 pour ("D")



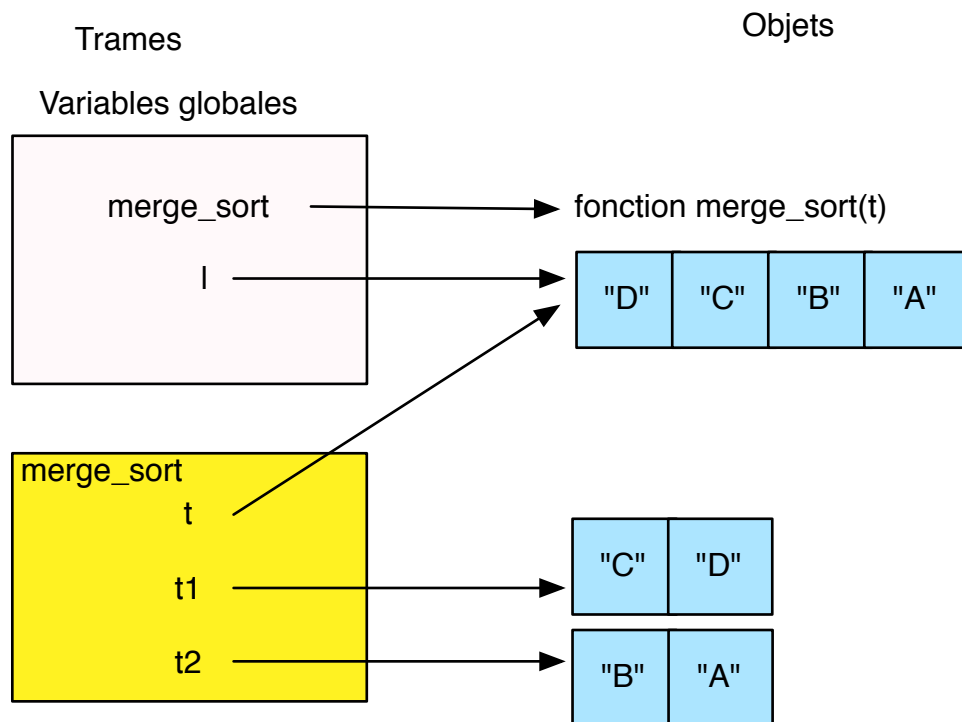
Etat dans l'instance de fonction de niveau 3 pour ("C")



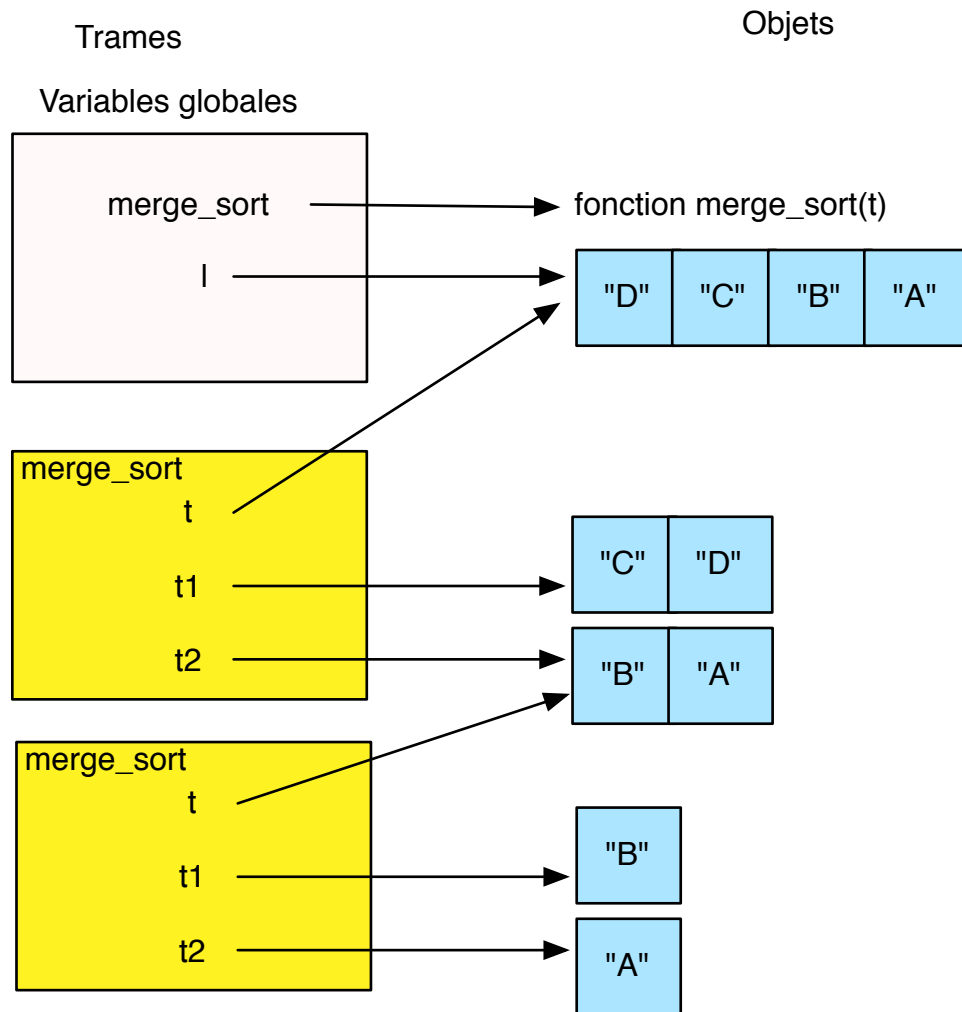
Etat dans l'instance de fonction de niveau 2 après la fusion de "D" et "C".



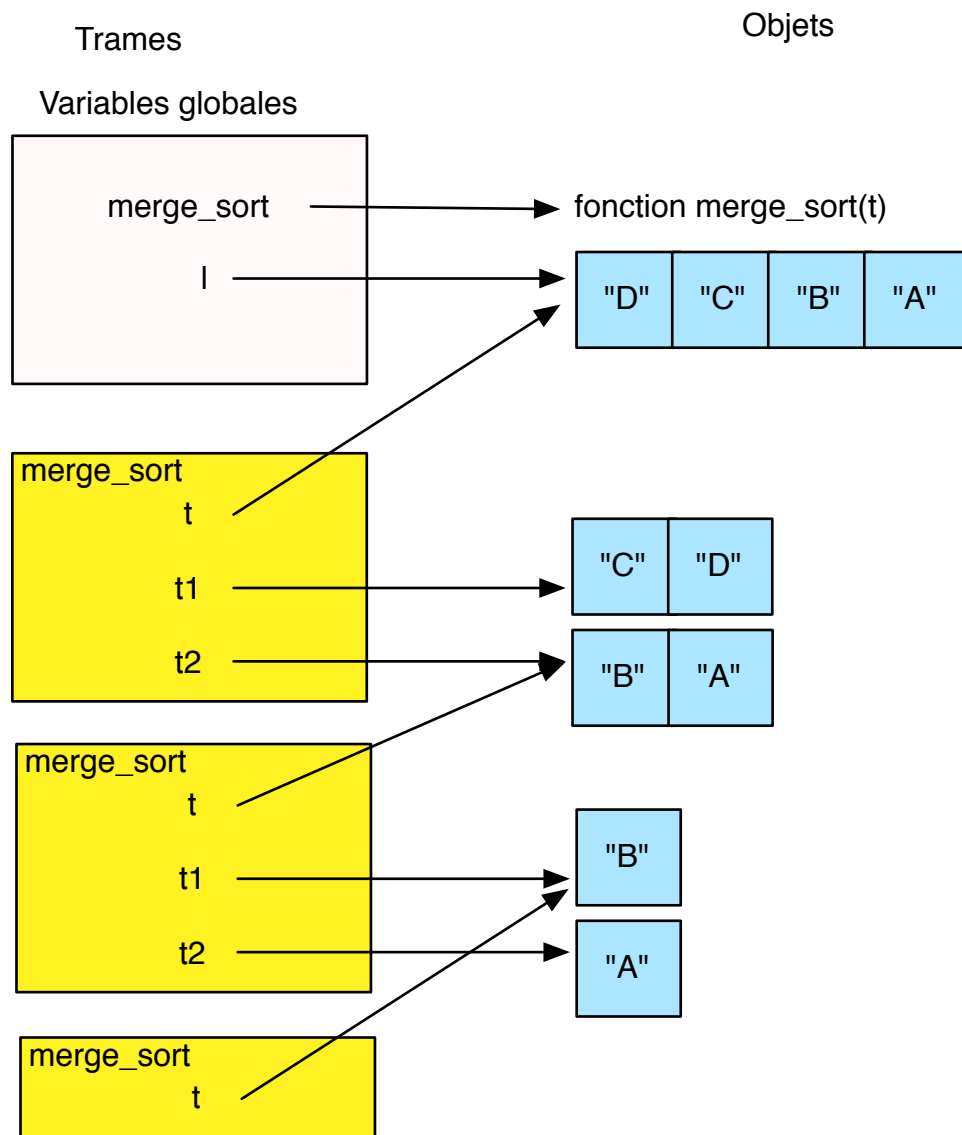
Modification de `t1` dans l'instance de fonction de niveau 1.



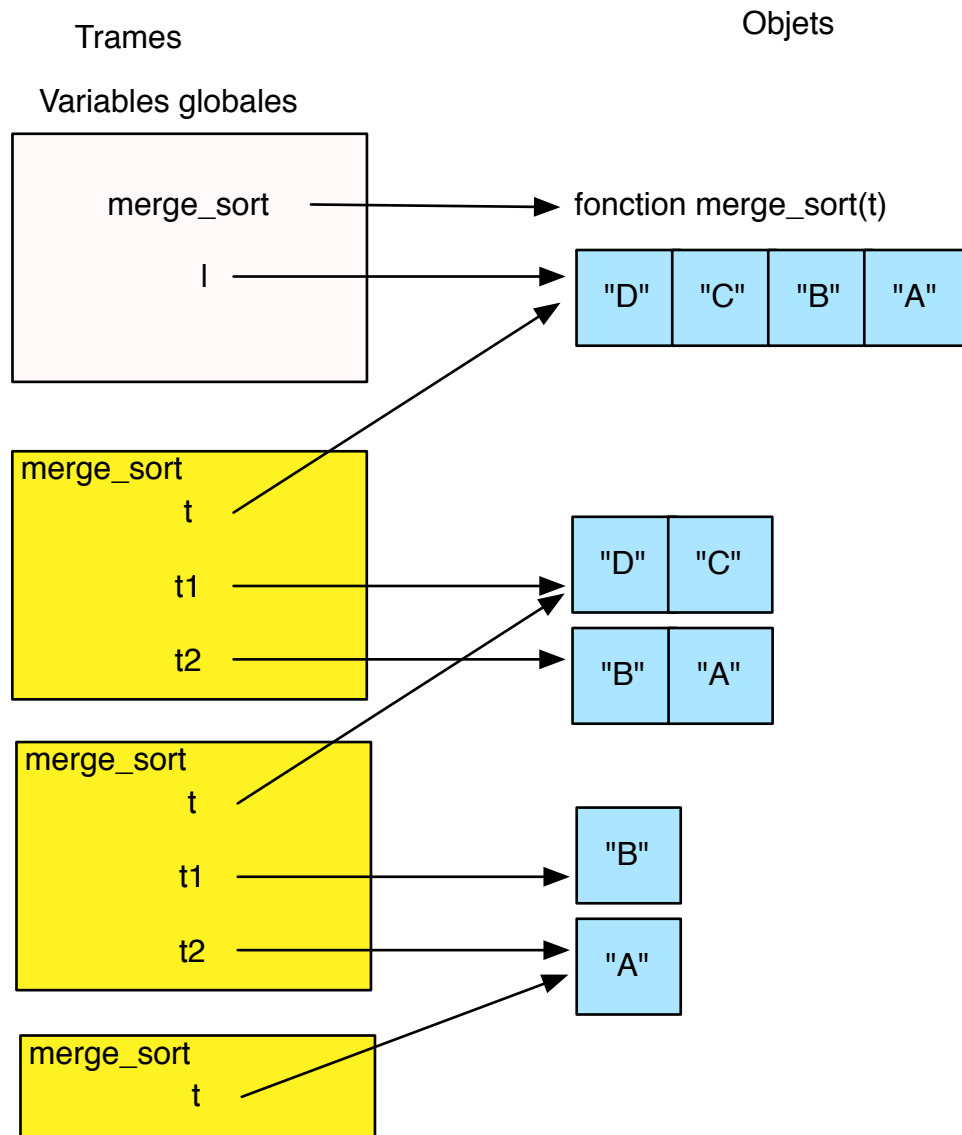
Etat dans l'instance de fonction de niveau 2 ("BA") avant les appels au niveau 3.



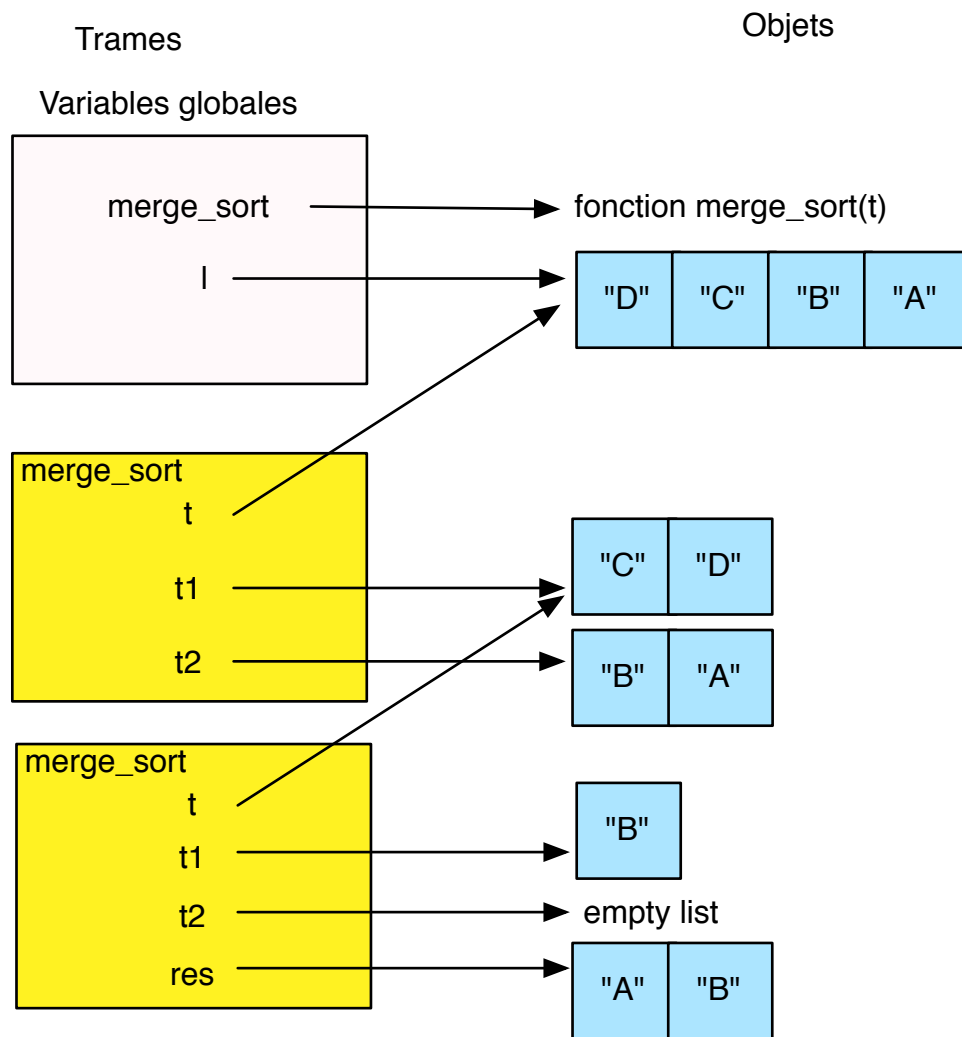
Etat dans l'instance de fonction de niveau 3 pour ("B")



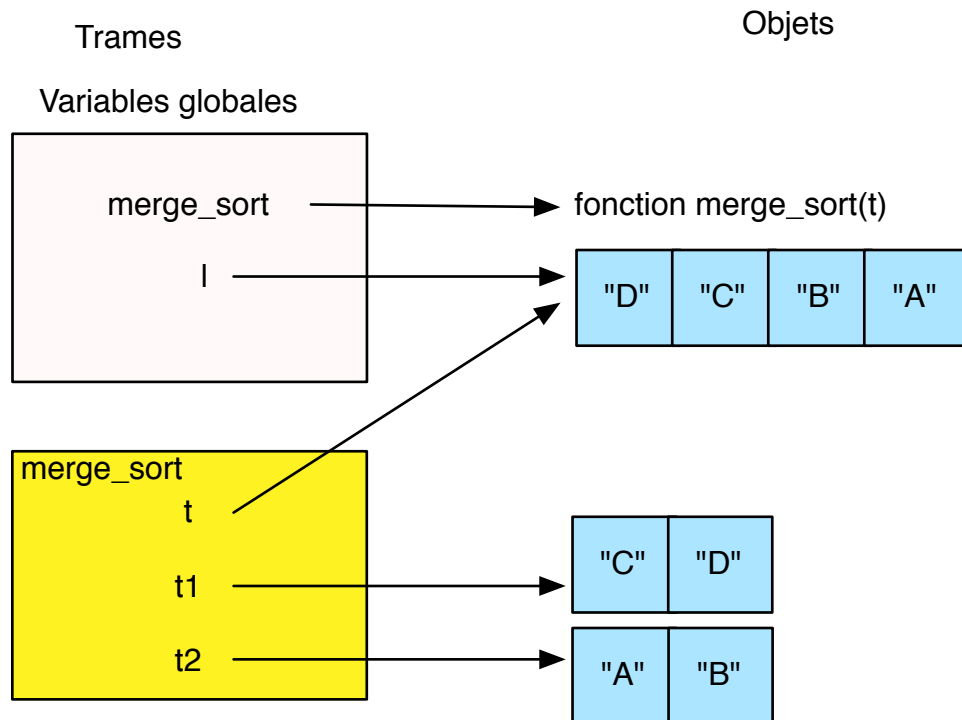
Etat dans l'instance de fonction de niveau 3 pour ("A")



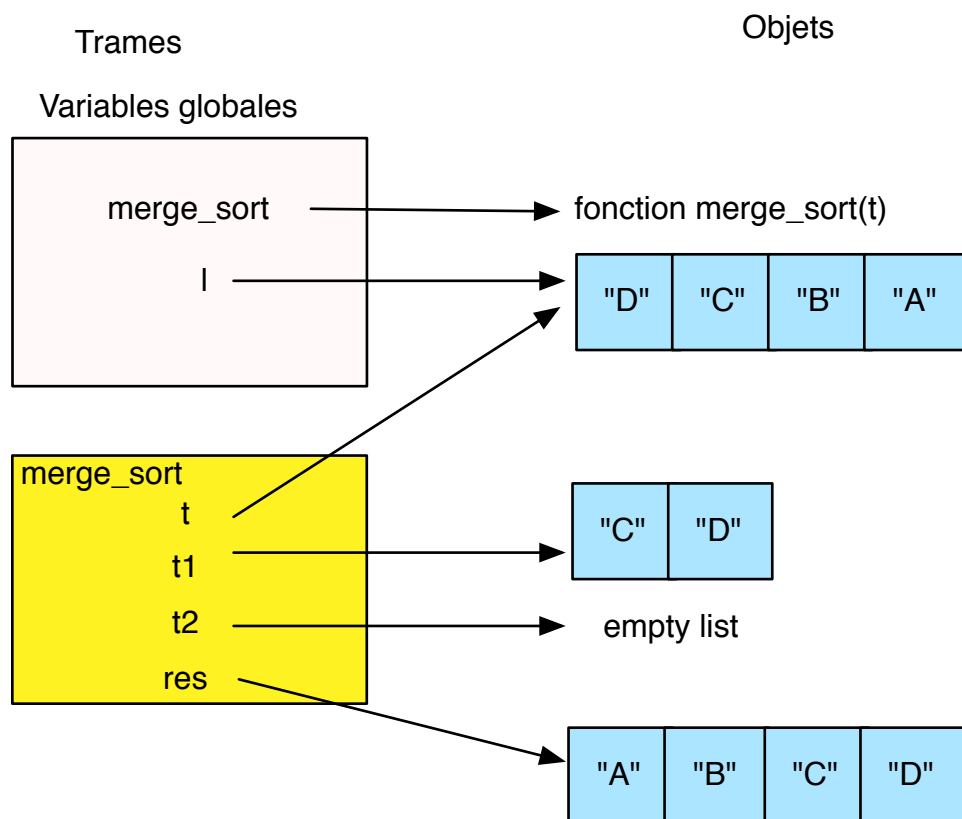
Etat dans l'instance de fonction de niveau 2 après la fusion de "B" et "A".



Modification de `t2` dans l'instance de fonction de niveau 1.



Après fusion de “CD” avec “AB” dans l’instance de niveau 1 de merge_sort ()



FICHIERS, PERSISTANCE ET EXCEPTIONS

See Also:

Lire le chapitres 9 du livre de Gérard Swinnen sur Python 3 ainsi que <http://docs.python.org/3/library/string.html> et <http://docs.python.org/3/library/exceptions.html>

12.1 Fichiers et persistance

Il est souvent intéressant de sauver des informations dans des fichiers qui continueront à exister après la fin de l'exécution du programme *python*. De telles informations sont appelées en informatique persistantes par opposition aux données volatiles créées en mémoire vive lors de l'exécution du programme et qui disparaissent en cours ou en fin de cette exécution. Les données persistantes sont généralement sauvées sur disque, sous forme soit de fichiers soit de bases de données.

Nous avons déjà vu, dans un chapitre précédent, qu'il était possible d'*ouvrir* un fichier texte

- en mode lecture ("read") et de lire les données qu'il contenait,
- ou en mode écriture ("write") et d'y écrire des résultats.

La liste des méthodes standard sur les fichiers est la suivante:

Instruction	Effet
<code>f=open('fichier') :</code>	ouvre 'fichier' en lecture
<code>f=open('fichier','w') :</code>	ouvre 'fichier' en écriture
<code>f=open('fichier','a') :</code>	ouvre 'fichier' en écriture en rajoutant après les données déjà présentes
<code>f.read() :</code>	retourne le contenu du fichier f
<code>f.readline() :</code>	lit une ligne
<code>f.readlines() :</code>	renvoie la liste des lignes de f
<code>f.write(s) :</code>	écrit la chaîne de caractères s dans le fichier f
<code>f.close() :</code>	ferme f

L'exemple suivant

- utilise un fichier `data.txt` de données, contenant des expressions à évaluer,

- utilise un fichier `result.txt` résultat qui va recevoir pour chaque expression lue, le texte de l'expression suivie de "=" et de la valeur obtenue grâce à la fonction `eval()` ..

```
d=open("data.txt")
r=open("result.txt", 'w')

for i in d:
    i = i.strip()
    r.write("{0} = {1}\n".format(i, eval(i)))

d.close()
r.close()
```

Note: la méthode `write` reçoit en paramètre un string qui sera imprimé. Comme nous l'avons déjà vu avec les `print`, la méthode `s.format()` peut être fort utile. Son principe général est d'examiner le string `s` et de remplacer les éléments entre accolades avec des numéros et des informations de formatage, par les paramètres donnés, formatés tel que spécifié.

12.1.1 Pickle: écriture d'un objet

Il est possible de sauvegarder des valeurs d'objets de n'importe quel type. Le problème est la phase d'encodage : on peut utiliser un fichier `texte` encore faut-il *traduire* la valeur de l'objet en string. Si cela semble facile pour des valeurs simples, l'exercice devient plus périlleux pour des séquences, ensembles ou dictionnaires.

Le module `pickle` contient une méthode `dump()` qui réalise cette traduction, appelée **sérialisation**.

Il suffit ensuite de sauver le texte produit par cette sérialisation en l'écrivant, par exemple, dans un fichier. Quand le programmeur veut *retrouver* la valeur sauvée, il devra la lire sur le fichier et faire le travail de traduction inverse, grâce à la méthode `load()`.

Donc, la méthode `pickle.dumps` prend un objet en paramètre et retourne une représentation de l'objet sous forme de chaîne. La chaîne obtenue n'est pas destinée à être compréhensible ou lue par l'humain. Par contre, la méthode `pickle.loads` prend une chaîne de ce type en paramètre et reconstitue un objet identique à celui donné à `pickle.dumps`.

Exemples :

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
'(lp0\nI1\naI2\naI3\na.'
>>> pickle.dumps({'a':3, 'b':8, 'c':9})
"(dp0\nS'a'\np1\nI3\nsS'c'\np2\nI9\nsS'b'\np3\nI8\ns."
>>> pickle.dumps('hello')
"S'hello'\np0\n."
>>> pickle.dumps(12)
'I12\n.'
>>> save_t = pickle.dumps(t)
```

```
>>> t2 = pickle.loads(save_t)
>>> t2
[1, 2, 3]
>>> t == t2
True
>>> t is t2
False
```

Les fonctions `dump` et `load` (sans “s”) prennent un fichier comme argument supplémentaire pour sauvegarder / récupérer un objet depuis le fichier.

```
>>> import pickle
>>> f = open('data.txt', 'w')
>>> d = {'a':3, 'b':8, 'c':9}
>>> pickle.dump(d, f)
>>> f.close()
>>> exit()
```

ensuite si on relance une nouvelle session,

```
>>> import pickle
>>> # d est sauvegardé dans data.txt
>>> f = open('data.txt')
>>> d = pickle.load(f)
>>> d
{'a': 3, 'c': 9, 'b': 8}
```

12.1.2 Shelve

Le pickling est un moyen simple de sauvegarder un objet (ou plusieurs objets séquentiellement dans le même fichier). Un autre moyen, plus souple, est d'utiliser le module `shelve` qui fournit une solution de persistance de type “base de données”. Concrètement, la base de donnée est écrite (automatiquement) dans un fichier; la méthode `shelve.open` crée un fichier de sauvegarde s'il n'existe pas, et retourne un objet de type `shelve`; un `shelve` fonctionne (presque) comme un dictionnaire. La plupart des opérations et méthodes des dictionnaires lui sont applicables; toute modification appliquée au `shelve` est (automatiquement) sauvegardée; la méthode `shelve.close` permet de fermer le `shelve`, comme pour un fichier.

```
>>> import shelve
>>> t = [0,1,2,3]
>>> p = (1, 2, 3)
>>> i = 12
>>> s = 'hello'
>>> db = shelve.open('data.db')
>>> db['liste'] = t
>>> db['point'] = p
>>> db['entier'] = i
>>> db['chaine'] = s
>>> print(db)
{'liste': [0, 1, 2, 3], 'chaine': 'hello', 'entier': 12, 'point': (1, 2, 3)}
```

```
>>> db.close()
>>> exit() # ferme la session
```

et relancer une session

```
>>> import shelve
>>> db = shelve.open('data.db')
>>> print(db['liste'])
[0, 1, 2, 3]
>>> db['chaine'] = 'bonjour'
>>> print(db)
{'liste': [0, 1, 2, 3], 'chaine': 'bonjour', 'entier': 12, 'point': (1, 2, 3)}
```

12.2 Exceptions

Les exceptions sont fréquentes, en particulier quand on manipule des fichiers.

```
>>> 3 / 0
ZeroDivisionError: integer division or modulo by zero
>>> t = list(range(4))
>>> t[4]
IndexError: list index out of range
>>> s = '2' + 2
TypeError: cannot concatenate 'str' and 'int' objects
>>> open('xyz.txt')
IOError: [Errno 2] No such file or directory: 'xyz.txt'
>>> open('/etc/passwd', 'w')
IOError: [Errno 13] Permission denied: '/etc/passwd'
>>> open('music')
IOError: [Errno 21] Is a directory: 'music'
```

12.2.1 Lancer une exception

Une exception indique que quelque chose d'exceptionnel s'est produit. Vous pouvez vous même lancer des exceptions.

```
def square_root(a, eps = 10**-9):
    if not isinstance(a,int) and not isinstance(a,float):
        raise TypeError('a must be an integer or a float')
    if a < 0.0:
        raise ValueError('a must be >= 0')
    x = 0.0
    approx = 1.0
    while (abs(approx - x) > eps):
        x = approx
        approx = (x + (a / x)) / 2.0
    return approx
```

```
>>> square_root(4)
2.0
>>> square_root(-2)
ValueError: a must be >= 0
>>> square_root('hello')
TypeError: a must be an integer or a float
```

La syntaxe pour lancer une exception est donc :

```
raise ExceptionType('message')
```

Les types d'exceptions les plus fréquentes que vous pouvez lancer sont :

Nom	Cause
ValueError:	Erreur de valeur (par ex., ne respecte pas les pré-conditions)
TypeError:	Erreur de type pour un paramètre
IndexError:	Indice hors bornes
IOError:	Erreur d'entrée / sortie (par ex. fichiers)

12.2.2 Gérer les exceptions

Jusqu'à présent, quand une exception se produit, le script s'interrompt brutalement. Comment gérer les exceptions pour éviter cette sortie brutale et effectuer du code spécifique quand une exception se produit ?

Solution 1 [utiliser des instructions conditionnelles pour] éviter les exceptions.

Exemple [avant d'ouvrir un fichier, vérifier qu'il existe, que] ce n'est pas un répertoire, etc. avec des fonctions comme `os.path.exists` ou `os.path.isdir`.

Inconvénients :

- code difficile à lire et à écrire car nombreux cas possibles;
- les tests peuvent être coûteux en temps de calcul;
- nécessite de penser aux cas exceptionnels, et de les gérer "a priori", plutôt que de se concentrer sur le code principal;
- le nombre d'exceptions possibles peut être très important. Par exemple, pour les fichiers, l'exception suivante suggère qu'il y a au moins 21 types d'erreurs possibles !

```
>>> open('music')
IOError: [Errno 21] Is a directory: 'music'
```

- comment être sûr de ne pas oublier un cas exceptionnel ?

Une meilleure solution serait de pouvoir écrire le code principal et demander à l'interpréteur d'essayer (try) de l'exécuter, et gérer les problèmes seulement s'ils se produisent.

C'est exactement ce que propose la syntaxe `try - except`.

Solution 2 (de loin préférée) : utiliser une instruction `try - except`.

La forme la plus simple d'une instruction `try - except` est la suivante.

```
try:
    # code principal
except:
    # code spécifique en cas d'exception
```

Comportement [le code principal est exécuté. Dès que celui-ci] produit une exception, il est interrompu et le code spécifique est exécuté. Si aucune exception ne s'est produite, le code spécifique n'est pas exécuté.

On dit qu'on rattrape (catch) une exception (dans une clause `except`).

Exemple :

```
try:
    print('Avant')
    raise ValueError('Test')
    print('Après')
except:
    print('Gestion exception')
print('Fin')
```

Résultat :

```
Avant
Gestion exception
Fin
```

Exemple :

```
try:
    i = int(input('Entier : '))
    print("Nous pouvons travailler avec", i)
except:
    print("Je ne peux travailler qu'avec un entier")
print("Fin du programme")
```

Si on entre un entier :

```
Entier : 2
Nous pouvons travailler avec 2
Fin du programme
```

Sinon :

```
Entier : hello
Je ne peux travailler qu'avec un entier
Fin du programme
```

Il est possible de spécifier dans la clause `except` le type d'exception que l'on veut rattraper.

Exemple :

```

ok = False
while not ok:
    try:
        x = int(input("Please enter a number: "))
        ok = True
    except ValueError:
        print("Oops! That was no valid integer. Try again...")
print("Now I'm sure that", x, "is a valid integer.")

```

Remarque : dans cet exemple, si une autre exception qu'une `ValueError` se produit, elle n'est pas rattrapée et le comportement sera celui habituel : le programme s'arrête brutalement.

On peut spécifier plusieurs clauses `except`, pour gérer différents types d'exceptions.

Exemple :

```

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError:
    print 'Cannot open myfile.txt'
except ValueError:
    print('Could not convert', s.strip(), 'to an integer.')
except:
    print('Unexpected error.')
    raise

```

Remarque : le dernier cas permet de relancer les exceptions qui n'ont pas été spécifiquement gérées (par ex. si ce code est encapsulé dans une fonction, l'appelant pourra alors gérer ces exceptions).

On peut spécifier un tuple d'exceptions à gérer dans une même clause `except`.

Exemple :

```

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except (IOError, ValueError):
    print('Cannot open file or cannot convert integer')

```

On peut placer un `else`, après les clauses `except`. Il sera exécuté si aucune exception n'est produite. C'est utile dans le cas où il faut exécuter du code seulement si l'instruction `try` n'a pas provoqué d'exceptions.

Exemple :

```

import sys
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:

```

```
    print('cannot open', arg)
else:
    print(arg, 'has', len(f.readlines()), 'lines')
    f.close()
```

Remarque : l'utilisation d'une clause `else` est ici meilleure que d'ajouter du code dans le `try` car cela évite d'attraper accidentellement une exception qui serait provoquée par une autre instruction que `open`.

Pour obtenir des informations sur une exception, on peut la faire suivre par le mot-clé `as` et le nom d'une variable. Cette variable possède comme valeur un objet de type exception. En affichant celle-ci, on obtient le message associé à l'exception.

Exemple :

```
try:
    f = open('myfile.txt')
except IOError as e:
    print('Cannot open myfile.txt:', e)
else:
    s = f.readline().strip()
    print(s)
```

Enfin, on peut ajouter une clause `finally` qui sera exécutée dans tous les cas (qu'il y ait une exception ou pas). Cela permet de réaliser des tâches de "clean-up" comme fermer un fichier qu'on savait ouvert avant le `try`, ou sauvegarder les données avant que le programme s'arrête suite à une exception, etc. La clause `finally` est exécutée juste avant de sortir du `try - except`, qu'une exception soit provoquée ou non.

Exemple :

```
try:
    raise KeyboardInterrupt
finally:
    print('Goodbye, world!')
```

donne:

```
Goodbye, world!
KeyboardInterrupt
```

Remarques : la commande `Ctrl-C` permet d'interrompre l'exécution d'un programme. En Python, une telle commande provoque un `KeyboardInterrupt`.

Exemple :

```
import time
print('This is a malicious script!')
print('Try to find the exit before the bomb explodes!')
try:
    for i in range(10):
        print(10-i, end=' ')
        if i % 2 == 0:
            print('tic')
```



```
        else:
            print('tac')
            time.sleep(1)
except KeyboardInterrupt:
    print('Well done! You have found the exit!')
else:
    print('BOOOM! You lose!')
finally:
    print('Goodbye!')
```

Exercice : Quels sont les messages que la fonction suivante va afficher (et dans quel ordre) si

- $x = 2$ et $y = 1$;
- $x = 2$ et $y = 0$;
- $x = '2'$ et $y = '1'$?

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```


CLASSES ET OBJETS

13.1 Objet et classe

Nous avons déjà vu que lors de l'exécution d'un programme, l'environnement d'exécution *Python* gère un espace mémoire, appelé *tas d'exécution* (*runtime heap*) et un autre espace mémoire, appelé *pile d'exécution* (*runtime stack*) qui, en particulier, à un instant donné, contient un espace de noms par instance locale d'une fonction en cours d'exécution.

Nous avons également vu qu'une variable est en fait un nom d'un objet; manipuler cet objet se fait grâce à cette variable qui l'identifie.

Tout objet *Python* x a :

- un identificateur unique et dont la valeur est constante tout au long de sa vie (donné par `id(x)`),
- un type (donné par `type(x)`),
- un contenu

Note:

- `a is b` est équivalent à `id(a) == id(b)`.
 - Deux objets dont les durées de vie n'ont pas d'intersection peuvent avoir le même identificateur.
-

On ne peut pas changer l'identité d'un objet ni son type (sauf exception). On peut changer le contenu des objets modifiables (mutables) sans changer leur identité ou leur type.

Un objet a donc un identificateur et un type et peut avoir un ou plusieurs noms. Il a également, en fonction du type de l'objet, une ou plusieurs *méthodes* qui permettent soit de *consulter* le contenu de l'objet soit, si l'objet est modifiable, de *modifier* son contenu.

Un objet a aussi des attributs (de données) qui sont manipulés par les méthodes `__setattr__` / `__getattr__` pour les attributs ou `__setitem__` / `__getitem__` pour les composants d'une séquence.

La section suivante explique comment *Python* permet au programmeur de définir des nouvelles classes d'objets et ainsi de créer et manipuler des objets de cette classe.

13.2 Définition de nouvelles classes

13.2.1 La notion de classe

On peut définir ses propres types en donnant une définition de classe. On peut donc voir une définition de classe comme la définition d'un nouveau type.

Définissons un nouveau type (classe) pour représenter des points dans le plan.

```
class Point(object):  
    """ représente un point dans le plan """
```

- le mot-clef `class` indique le début d'une définition de classe;
- on donne ensuite le nom de la classe (par convention, avec une majuscule);
- entre parenthèses, on précise qu'un "élément" de classe `Point` sera une "sorte" d'object, qui est une classe prédéfinie.

Pour l'instant notre modèle (classe) est très peu précis. On a juste donné un nom et un docstring.

```
>>> print(Point)  
<class '__main__.Point'>  
>>> help(Point)  
Help on class Point in module __main__:  
  
class Point(builtins.object)  
|   représente un point dans le plan  
|  
|   Data descriptors defined here:  
|  
|   __dict__  
|       dictionary for instance variables (if defined)  
|  
|   __weakref__  
|       list of weak references to the object (if defined)  
(END)
```

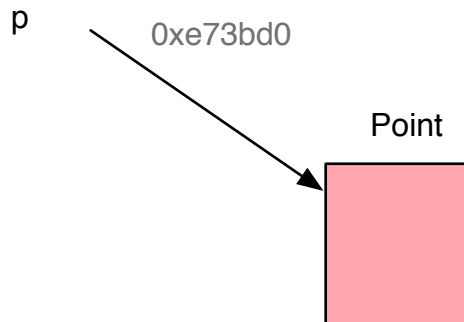
13.2.2 La notion d'objets

Cependant, on peut déjà créer des objets de la classe `Point`. Pour ce faire, on appelle `Point` comme si c'était une fonction.

```
>>> p = Point()  
>>> print(p)  
<__main__.Point object at 0xe73bd0>
```

- la valeur de retour est une référence vers un objet de type `Point`, que l'on assigne à la variable `p`;
- créer un objet est une *instanciation*, et l'objet est une *instance* de la classe `Point`;

- quand on affiche une instance, *Python* nous dit à quelle classe elle appartient et où elle est stockée en mémoire (sous la forme d'une adresse représentée par un nombre hexadécimal).



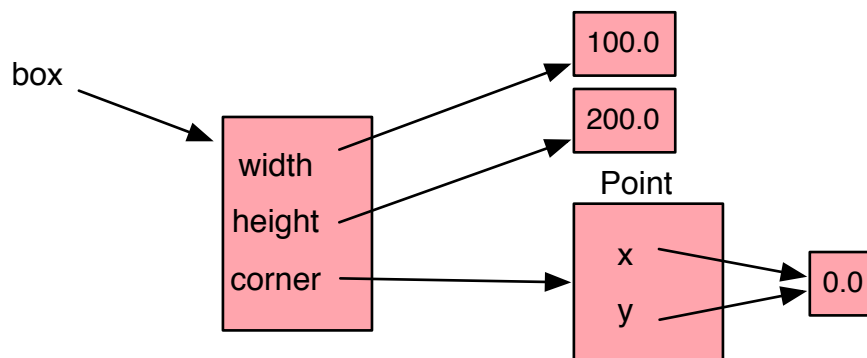
13.2.3 Attributs

Une des propriétés des objets est qu'il possède des **attributs**. Un attribut est une valeur.

Par exemple, pour représenter un point dans le plan, on a besoin de deux attributs (coordonnées x et y).

Pour assigner un attribut à un objet, on peut utiliser la notation point.

```
>>> p.x = 2.0
>>> p.y = 4.0
```



On accède aux attributs d'un objet également via la notation point.

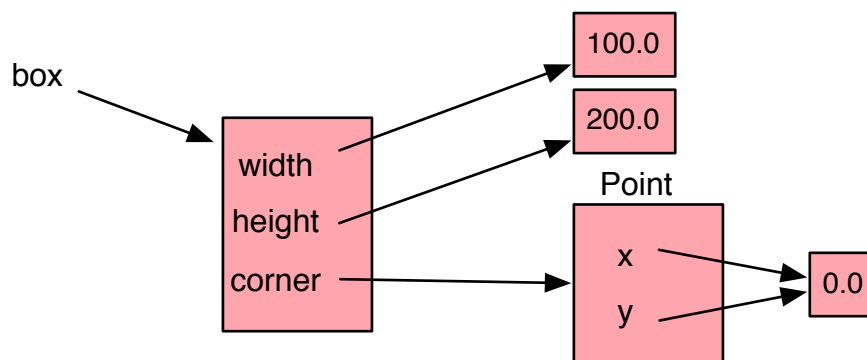
$p.x$ peut être lu comme “aller dans l'objet référencé par p et récupérer la valeur de son attribut x ”

```
>>> import math
>>> print(p.x)
2.0
>>> y = p.y
>>> print(y)
4.0
>>> print("{0}, {1}".format(p.x, p.y))
(2.0, 4.0)
```

```
>>> dist_from_orig = math.sqrt(p.x**2 + p.y**2)
>>> print(dist_from_orig)
4.472135955
```

Un attribut peut être une référence vers un autre objet.

```
>>> class Rectangle(object):
...     """ représente un rectangle dans le plan
...         attributs: width, height, corner
...         (coin inférieur gauche, Point)
...     """
...
>>> box = Rectangle()
>>> box.width = 100.0
>>> box.height = 200.0
>>> box.corner = Point()
>>> box.corner.x = 0.0
>>> box.corner.y = 0.0
```



13.2.4 Objet comme valeur de retour

Une fonction peut créer un objet et retourner la référence vers celui-ci.

```
def find_center(r):
    c = Point()
    c.x = r.corner.x + r.width / 2.0
    c.y = r.corner.y + r.height / 2.0
    return c
```

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

13.2.5 Copie d'objets

Un objet passé en argument peut être modifié par une fonction. On a vu que cela peut provoquer des problèmes dans certains cas (par exemple pour les listes non simples).

On peut copier un objet pour éviter l'*aliasing* grâce au module `copy`.

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0
>>> p2 = p1
>>> p2.y = 7.0
>>> print_point(p1)
(3, 7)
>>> p1 is p2
True
>>> import copy
>>> p2 = copy.copy(p1)
>>> p2.y = 9.0
>>> print_point(p1)
(3, 7)
>>> print_point(p2)
(3, 9)
>>> p1 is p2
False
```

Notons que :

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0
>>> p2 = copy.copy(p1)
```

`p1 == p2` retourne `False` !

En effet, pour les objets, le comportement par défaut de `==` est le même que l'opérateur `is`. Mais ce comportement pourra être modifié (dans la définition de classe).

Soit :

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
```

Mais `box2.corner is box.corner` renvoie `True` car la fonction `copy.copy` réalise une "shallow copy", c-à-d qu'elle copie les références contenues dans les attributs mutables.

Rappel [la fonction `copy.deepcopy` réalise une "deep copy",] c-à-d qu'elle copie "également" les objets référencés dans les attributs mutables (et ceux référencés par ceux-ci, etc.).

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

Les objets référencés par `box` et `box3` sont maintenant complètement séparés (et indépendants).

13.2.6 Méthodes

Les attributs d'un objet modélisent son état, ses particularités dans la famille des objets du même type.

Exemples:

- un point a comme coordonnées (2,3); un autre (4,5), mais ce sont deux points dans le plan.
- un rectangle a une largeur de 10, une hauteur de 20, et un coin inférieur droit en (3,7). Un autre sera plus grand, ou plus à droite, mais il reste un rectangle.
- un temps dans la journée correspond à la manière dont nous décrivons un temps donné, c-à-d, une heure, une minute et une seconde.

Mais un objet possède d'autres propriétés que ses attributs : les *méthodes* associées à son type (sa classe).

Une *méthode* est une fonction qui est définie pour un type donné et qui s'applique sur les objets de ce type.

Intuitivement, une méthode est une action appliquée sur un objet (et peut avoir besoin d'autres entrées que l'objet sur lequel elle est appliquée, et peut produire également des sorties).

Exemple: Pour manipuler des heures nous pouvons définir une classe Time :

```
class Time(object):  
    """ represente un temps (heure).  
        attributs: hour, minute, second  
    """
```

Nous pourrions définir une méthode display associée à le classe Time :

```
>>> class Time(object):  
    """ represente un temps (heure).  
        attributs: hour, minute, second  
    """  
    def display(self):  
        print("{0}:{1}:{2}".format(self.hour, self.minute, self.second))  
  
>>> start = Time()  
>>> start.hour = 9  
>>> start.minute = 45  
>>> start.second = 0  
>>> start.display()  
09:45:00
```

Définir une méthode revient simplement à donner sa définition à l'intérieur de la définition de la classe. Pour accéder à l'objet sur lequel la méthode sera invoquée, on utilise la première variable nommée self (par convention).

Une méthode destinée à être invoquée sur un objet possède toujours cet objet comme premier paramètre; quand on invoque la méthode sur un objet, il ne faut pas donner l'argument (self); ce type de méthodes est appelé une méthode d'instance;

Note: il existe d'autres types de méthodes (de classes et statiques) qui ne seront pas vues dans ce cours.

Notons que l'on a ici une ébauche de programmation orientée-objets :

```
>>> start.display()
09:45:00
```

où les objets sont les agents actifs.

Supposons que l'on veuille avoir une méthode qui additionne un temps à un autre. Un des deux objets sera l'objet sur lequel la méthode sera invoquée et qui sera modifié, l'autre sera donné en paramètre.

Pour cela nous allons d'abord écrire une méthode `to_sec` qui traduit un temps en secondes. Inversément écrivons une méthode `update` qui prend un nombre de secondes en arguments et met à jour l'objet `Time` en transformant ce nombre en temps.

```
class Time(object):
    # ...
    def to_sec(self):
        mins = self.hour * 60 + self.minute
        secs = mins * 60 + self.second
        return secs

    def update(self, secs):
        mins, self.second = divmod(secs, 60)
        self.hour, self.minute = divmod(mins, 60)

    def add(self, other):
        secs = self.to_sec() + other.to_sec()
        self.update(secs)
```

On pourra alors par exemple avoir:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 0
>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0
>>> start.display()
09:45:00
>>> duration.display()
01:35:00
>>> duration.to_sec()
5700
>>> duration.update(5760)
>>> duration.display()
```

```
01:36:00
>>> start.add(duration)
>>> start.display()
11:21:00
```

13.2.7 La méthode init

La méthode `init` (`__init__`) est une méthode spéciale qui est invoquée automatiquement quand un objet est créé. Cela permet d'ajouter des arguments (par défaut) quand on crée un objet.

```
class Time(object):
    # ...
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
    # ...
```

On pourra ainsi avoir le code:

```
>>> t1 = Time()
>>> t2 = Time(3)
>>> t3 = Time(4, 12)
>>> t4 = Time(18, 35, 17)
>>> t1.display()
00:00:00
>>> t2.display()
03:00:00
>>> t3.display()
04:12:00
>>> t4.display()
18:35:17
```

13.2.8 La méthode str

La méthode `str` (`__str__`) est une méthode spéciale qui est censée retourner une représentation de l'objet sous forme de chaîne de caractères. Lors d'une instruction `print`, *Python* appelle automatiquement cette méthode.

```
class Time(object):
    # ...
    def __str__(self):
        s = "{0:2d}:{1:2d}:{2:2d}".format(self.hour, self.minute, self.second)
        return s
    # ...
```

On pourra ainsi avoir le code:

```
>>> t = Time(8,30)
>>> print(t)
08:30:00
>>> str(t)
'08:30:00'
```

- la méthode `display` devient inutile.
- `__init__` et `__str__` devraient toujours être présentes dans vos définitions de classes (en général, on commence par ces méthodes).

13.2.9 Surcharge d'opérateur

On peut également définir des opérateurs sur les objets. Par exemple, l'opérateur `+` peut être défini en écrivant une méthode spéciale `__add__` pour la classe `Time`.

On parle de *surcharge des opérateurs* : adapter le comportement d'un opérateur à un type défini par le programmeur.

```
class Time(object):
    # ...
    def __add__(self, other):
        res = Time()
        secs = self.to_sec() + other.to_sec()
        res.update(secs)
        return res
    # ...
```

On pourra ainsi avoir le code:

```
>>> t1 = Time(9)
>>> t2 = Time(1,30)
>>> t3 = t1 + t2
>>> print(t3)
10:30:00
```

Pour chaque opérateur présent en *Python*, il y a une méthode spéciale qui correspond et qui permet de surcharger l'opérateur ! (Voir: <http://docs.python.org/3/reference/datamodel.html#special-method-names>)

Time + int : pour le moment l'opérateur `+` opère sur deux objets `Time`. Nous voudrions pouvoir également pouvoir ajouter un nombre entier (secondes) à ces objets.

```
class Time(object):
    # ...
    def __add__(self, other):
        res = Time()
        secs = self.to_sec()
        if isinstance(other, Time):
            secs += other.to_sec()
        elif isinstance(other, int):
            secs += other
```

```
    else:
        raise NotImplemented('op + only for Time and int')
    res.update(secs)
    return res
# ...
```

On pourra ainsi avoir le code:

```
>>> t1 = Time(9)
>>> t2 = t1 + 600
>>> print(t2)
09:10:00
>>> t3 = t1 + t2
>>> print(t3)
18:10:00
```

Problème : malheureusement, cette addition n'est pas commutative. Si l'entier est le premier opérateur, on obtient une erreur.

```
>>> 1200 + t3
TypeError: unsupported operand type(s) for +: 'int' and 'Time'
```

Le problème vient du fait qu'au lieu de demander à un objet Time d'ajouter un entier, on fait le contraire, et on ne peut pas modifier les objets de type entiers pour qu'ils connaissent les objets Time.

Solution : écrire la méthode spéciale `__radd__` (right-side add). Cette méthode est automatiquement invoquée quand un objet Time est le membre droit d'un opérateur + (et que la méthode `__add__` n'existe pas ou retourne `NotImplemented` sur le membre gauche).

```
class Time(object):
    # ...
    def __radd__(self, other):
        return self.__add__(other)
    # ...
```

On pourra ainsi avoir le code:

```
>>> t1 = Time(8)
>>> t2 = 1200 + t1
>>> print(t2)
08:20:00
```

Note: les méthodes spéciales peuvent être également invoquées “à la main”, comme n'importe quelle autre méthode.

13.2.10 Polymorphisme

On a déjà vu qu'une fonction peut s'appliquer sur différents types d'objets, pour peu que le corps de la fonction applique des opérateurs disponibles pour ce type d'objet. On appelle cette

caractéristique le polymorphisme. Cela signifie par exemple que l'on peut utiliser la fonction `sum` sur nos objets `Time` !

```
>>> sum(range(10))
45
>>> t = [Time(1,30), Time(2), Time(3,45,30)]
>>> total_time = sum(t)
>>> print(total_time)
07:15:30
```

13.3 Programmation orientée objet

Des langages comme *Python*, *C++* ou *Java* permettent de définir des classes, d'instancier des objets et de les manipuler.

Cette partie de la syntaxe n'est pas nécessaire (on pourrait tout faire rien qu'avec des fonctions), mais elle permet une programmation intuitive, concise et réutilisable : la *programmation orientée objet*.

Nous n'avons couvert qu'une toute petite partie des concepts orientés objets. Il existe une série de techniques très intéressantes et permettant d'accentuer encore les avantages précités.

Les cours de programmation ou d'algorithmiques suivants couvriront bien plus cette matière.

QUELQUES MOTS SUR LES LANGAGES DE PROGRAMMATION

Note: La plupart des sections de ce chapitre proviennent des différentes sources listées ci-dessous:

- http://en.wikipedia.org/wiki/History_of_programming_languages (consultation 4 décembre 2012)
 - http://fr.wikipedia.org/wiki/Histoire_des_langages_de_programmation (consultation 4 décembre 2012)
 - http://en.wikipedia.org/wiki/Programming_language (consultation 4 décembre 2012)
 - http://fr.wikipedia.org/wiki/Langage_de_programmation (consultation 4 décembre 2012)
 - <http://www.computerhistory.org/> (consultation 5 décembre 2012)
 - <http://histoire.info.online.fr/> (consultation 5 décembre 2012)
 - <http://www.histoire-informatique.org/> (consultation 5 décembre 2012)
 - Ancien syllabus du cours (langage C++)
-

14.1 Introduction

Un langage de programmation est une notation artificielle, destinée à exprimer des algorithmes et produire des programmes. D'une manière similaire à une langue naturelle, un langage de programmation est fait d'un alphabet, un vocabulaire, des règles de grammaire (syntaxe), et des significations (sémantique). Les langages de programmation servent à décrire les structures des données qui seront manipulées par l'ordinateur, et à indiquer comment sont effectuées les manipulations, selon quels algorithmes. Ils servent de moyens de communication par lesquels le programmeur communique avec l'ordinateur, mais aussi avec d'autres programmeurs; les programmes étant d'ordinaire écrits, lus, compris et modifiés par une communauté.

Un langage de programmation est mis en oeuvre par un traducteur automatique: compilateur ou interpréteur.

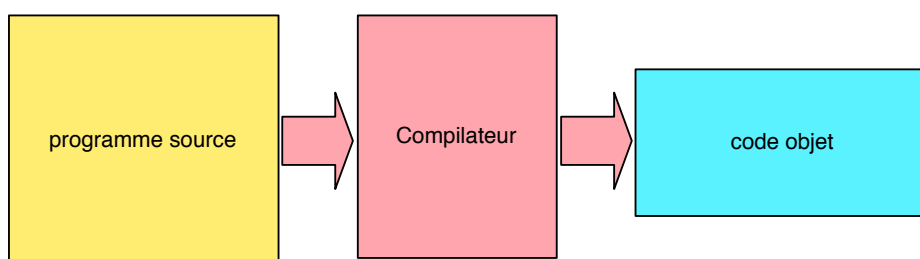
Les langages de programmation offrent différentes possibilités d'abstraction, et une notation proche de l'algèbre, permettant de décrire de manière concise et facile à saisir les opérations de manipulation de données et l'évolution du déroulement du programme en fonction des situations. La possibilité d'écriture abstraite libère l'esprit du programmeur d'un travail superflu, et lui permet de se concentrer sur des problèmes plus avancés.

Chaque langage de programmation reflète un ou plusieurs paradigmes, un ensemble de notions qui orientent le travail de réflexion du programmeur, sa technique de programmation et sa manière d'exprimer le fruit de ses réflexions dans un langage de programmation.

Les premiers langages de programmation ont été créés dans les années 1950. De nombreux concepts ont été lancés par un langage, puis améliorés et étendus dans les langages suivants. La plupart du temps la conception d'un langage de programmation a été fortement influencée par l'expérience acquise avec les langages précédents

14.1.1 Compilateur

Un compilateur est un programme informatique qui transforme un code source écrit dans un langage de programmation (le langage source) en un autre langage informatique (le langage cible).



Pour qu'il puisse être exploité par la machine, le compilateur traduit le code source, écrit dans un langage de haut niveau d'abstraction, facilement compréhensible par l'humain, vers un langage de plus bas niveau, un langage d'assemblage ou langage machine. Dans le cas de langage semi-compilé (ou semi-interprété), le code source est traduit en un langage intermédiaire, sous forme binaire (code objet ou bytecode), avant d'être lui-même interprété ou compilé.

Un compilateur effectue les opérations suivantes : analyse lexicale, pré-traitement (préprocesseur), analyse syntaxique (parsing), analyse sémantique, génération de code intermédiaire, optimisation de code et génération de code final.

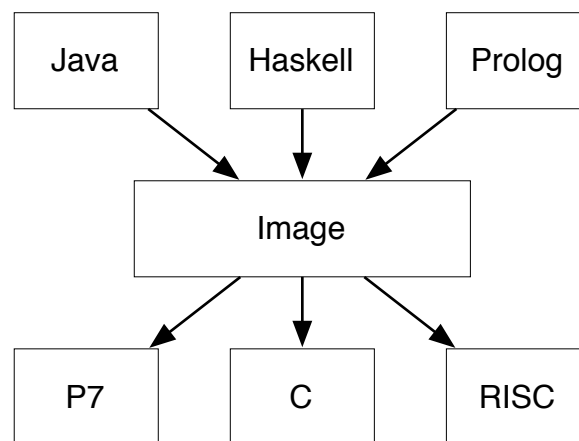
Quand le programme compilé (code objet) peut être exécuté sur un ordinateur dont le processeur ou le système d'exploitation est différent de celui du compilateur, on parle de compilateur croisé (cross-compiler).

Structure et fonctionnement

La tâche principale d'un compilateur est de produire un code objet correct qui s'exécutera sur un ordinateur. La plupart des compilateurs permettent d'optimiser le code, c'est-à-dire qu'ils vont chercher à améliorer la vitesse d'exécution, ou réduire l'occupation mémoire du programme.

Un compilateur fonctionne par analyse-synthèse : au lieu de remplacer chaque construction du langage source par une suite équivalente de constructions du langage cible, il commence par analyser le texte source pour en construire une représentation intermédiaire (appelée image) qu'il traduit à son tour en langage cible.

On sépare le compilateur en au moins deux parties : une partie avant (ou frontale - front-end) qui lit le texte source et produit la représentation intermédiaire (appelée également image); et une partie arrière (ou finale - back-end), qui parcourt cette représentation pour produire le code cible. Dans un compilateur idéal, la partie avant est indépendante du langage cible, tandis que la partie arrière est indépendante du langage source.



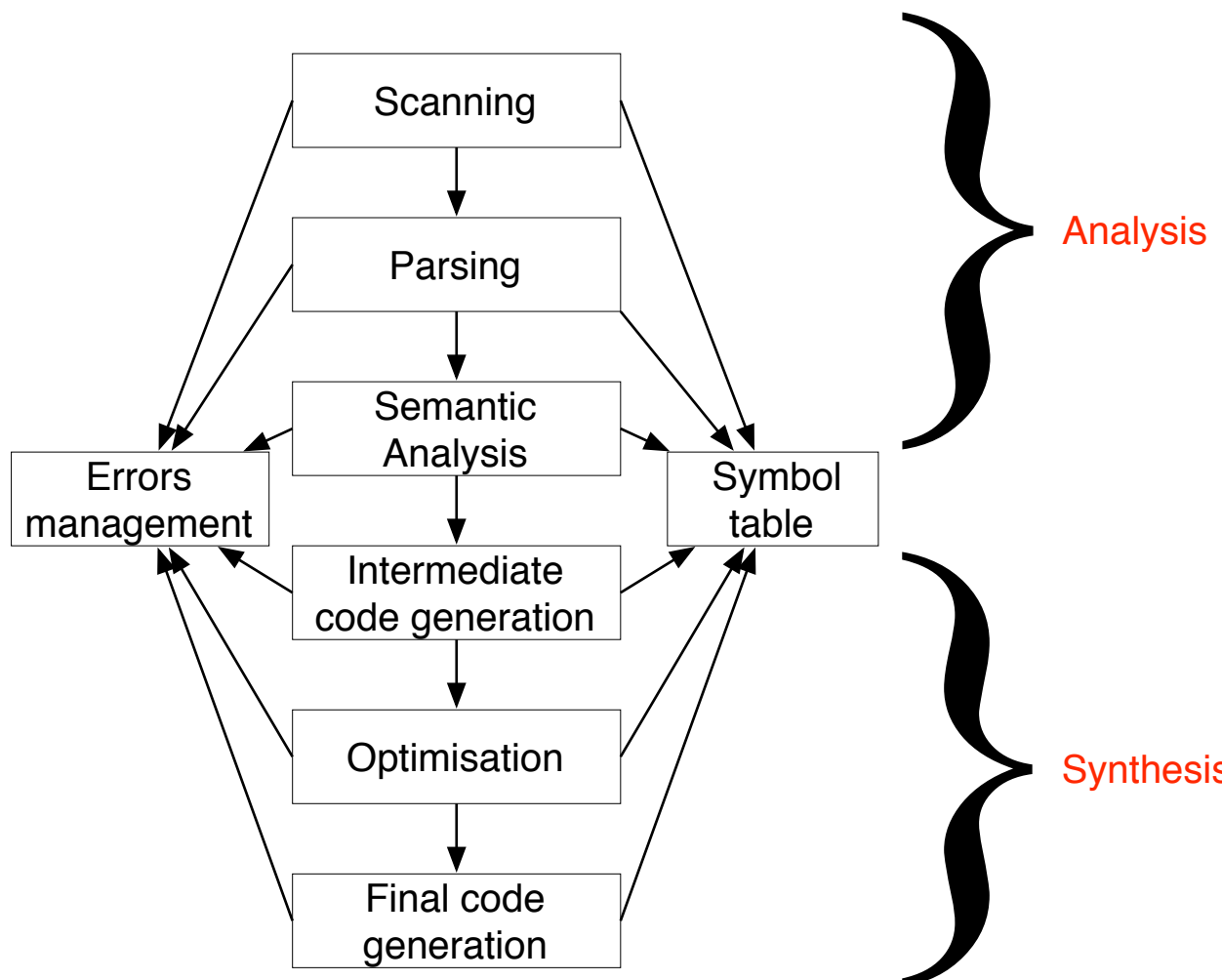
Certains compilateurs effectuent des traitements substantiels sur la partie intermédiaire, devenant une partie centrale à part entière, indépendante à la fois du langage source et de la machine cible. On peut ainsi écrire des compilateurs pour toute une gamme de langages et d'architectures en partageant la partie centrale, à laquelle on attache une partie avant par langage et une partie arrière par architecture.

Les étapes de la compilation incluent :

- le prétraitement, nécessaire pour certaines langues comme C, qui prend en charge la substitution de macro et de la compilation conditionnelle. Généralement, la phase de prétraitement se produit avant l'analyse syntaxique ou sémantique.
- l'analyse lexicale (scanning), qui découpe le code source en petits morceaux appelés jetons (tokens). Chaque jeton est une unité atomique unique de la langue (unités lexicales ou lexèmes), par exemple un mot-clé, un identifiant ou un symbole. Le logiciel qui effectue une analyse lexicale est appelé un analyseur lexical ou un scanner. Deux outils classiques permettent de construire plus aisément des scanners : lex et flex.
- l'analyse syntaxique implique l'analyse de la séquence de jetons pour identifier la structure syntaxique du programme. Cette phase s'appuie généralement sur la construction d'un arbre d'analyse ; on remplace la séquence linéaire des jetons par une structure en arbre construite selon la grammaire formelle qui définit la syntaxe du langage. Par exemple, une condition est toujours suivie d'un test logique (égalité, comparaison...). L'arbre d'analyse est souvent modifié et amélioré au fur et à mesure de la compilation. Yacc et GNU Bison sont les outils les plus utilisés pour construire des analyseurs syntaxiques.
- l'analyse sémantique est la phase durant laquelle le compilateur ajoute des informations

sémantiques à l'arbre d'analyse et construit la table des symboles. Cette phase fait un certain nombre de contrôles: vérifie le type des éléments (variables, fonctions, ...) utilisés dans le programme, leur visibilité, vérifie que toutes les variables locales utilisées sont initialisées, L'analyse sémantique nécessite habituellement un arbre d'analyse complet, ce qui signifie que cette phase fait suite à la phase d'analyse syntaxique, et précède logiquement la phase de génération de code ; mais il est possible de replier ces phases en une seule passe.

- la transformation du code source en code intermédiaire ;
- l'application de techniques d'optimisation sur le code intermédiaire : c'est-à-dire rendre le programme « meilleur » selon son usage;
- la génération de code avec l'allocation de registres et la traduction du code intermédiaire en `C` ou en assembleur.



L'analyse lexicale, syntaxique et sémantique, le passage par un langage intermédiaire et l'optimisation forment la partie frontale. La génération de code ¹ constitue la partie finale.

Ces différentes étapes font que les compilateurs sont toujours l'objet de recherches.

¹ avec l'édition de liens,

14.1.2 Interpréteur

Un interpréteur est un outil informatique ayant pour tâche d'analyser, de traduire et d'exécuter un programme écrit dans un langage informatique. De tels langages sont dits langages interprétés.

L'interpréteur est capable de lire le code source d'un langage sous forme de script, habituellement un fichier texte, et d'en exécuter les instructions (une à la fois) après une analyse syntaxique du contenu. Cette interprétation conduit à une exécution d'action ou à un stockage de contenu.

Principe

L'interprétation consiste en l'exécution dynamique du programme par un autre programme appelé interpréteur, plutôt que sur sa conversion (compilation) en un autre langage (par exemple le langage machine) ; ainsi la "traduction" et l'exécution sont simultanées.

Le cycle d'un interpréteur est le suivant :

- lire et analyser une instruction (ou expression) ;
- si l'instruction est syntaxiquement correcte, l'exécuter (ou évaluer l'expression) ;
- passer à l'instruction suivante.

Ainsi, contrairement au compilateur, l'interpréteur exécute les instructions du programme (ou en évalue les expressions), au fur et à mesure de leur lecture pour interprétation. Du fait de cette phase sans traduction préalable, l'exécution d'un programme interprété est généralement plus lente que le même programme compilé.

La plupart des interpréteurs n'exécutent plus la chaîne de caractères représentant le programme, mais une forme interne, telle qu'un arbre syntaxique.

En pratique, il existe souvent des mélanges entre interpréteurs et compilateurs.

L'intérêt des langages interprétés réside principalement dans la facilité de programmation et dans la portabilité. Les langages interprétés facilitent énormément la mise au point des programmes car ils évitent la phase de compilation, souvent longue, et limitent les possibilités de bogues. Il est en général possible d'exécuter des programmes incomplets, ce qui facilite le développement rapide d'applications ou de prototypes d'applications.

Ainsi, le langage BASIC fut le premier langage interprété à permettre au grand public d'accéder à la programmation, tandis que le premier langage de programmation moderne interprété est Lisp.

La portabilité permet d'écrire un programme unique, pouvant être exécuté sur diverses plateformes sans changements, pourvu qu'il existe un interpréteur spécifique à chacune de ces plateformes matérielles.

Les inconvénients d'une interprétation plutôt qu'une compilation sont la lenteur lors de l'exécution, comme déjà évoqué, mais aussi l'absence de contrôle préalable sur la structure du programme et les types des éléments manipulés avant le début de son exécution.

Un certain nombre de langages informatiques sont aujourd'hui mis en oeuvre au moyen d'une machine virtuelle applicative. Cette technique est à mi-chemin entre les interpréteurs tels que décrits ici et les compilateurs. Elle offre la portabilité des interpréteurs avec une bonne efficacité. Par exemple, des portages de Java, Lisp, Scheme, Ocaml, Perl (Parrot), Python, Ruby, Lua, C, etc. sont faits via une machine virtuelle.

Note: Lors de la conception d'un programme dans un langage interprété, étant donné l'absence d'un compilateur qui contrôle les types, la visibilité, ..., les techniques de conception insistent fortement sur la notion de **test unitaire** qui teste si possible l'entièreté du code d'une fonction ou d'une classe, à la fois au niveau fonctionnalité qu'en testant l'ensemble des lignes de code écrites, avant l'intégration de ce code dans le reste du programme.

Notons cependant que tester ou vérifier la logique d'un code est généralement plus complexe et doit se faire tant avec les langages interprétés que compilés.

Historique

Avec l'apparition du langage compilé Pascal et de compilateurs commerciaux rapides comme Turbo Pascal, les langages interprétés connurent à partir du milieu des années 1980 un fort déclin. Trois éléments changèrent la donne dans les années 1990 :

- avec la nécessité d'automatiser rapidement certaines tâches complexes, des langages de programmation interprétés (en fait, semi-interprétés) de haut niveau comme, entre autres, Tcl, Ruby, Perl ou Python se révélèrent rentables ;
- la puissance des machines, qui doublait tous les dix-huit mois en moyenne (selon la loi de Moore), rendait les programmes interprétés des années 1990 d'une rapidité comparable à celle des programmes compilés des années 1980 ;
- il est bien plus rapide de faire évoluer un programme interprété. Or la vague Internet demandait une réponse très rapide aux nouveaux besoins du marché. amazon.com fut, dans sa première version, développé largement en Perl. Smalltalk permettait un prototypage très rapide d'applications.

14.1.3 Runtime

Un *runtime* est l'environnement d'exécution du programme qui utilise un ensemble de bibliothèques logicielles pour mettre en oeuvre le langage de programmation, permettant d'effectuer des opérations simples telles que copier des données, ou des opérations beaucoup plus complexes telles que le travail de *garbage collection* (ramasse-miettes).

Lors de la traduction d'un programme vers le langage machine les opérations simples sont traduites en les instructions correspondantes en langage machine tandis que les opérations complexes sont traduites en des utilisations de fonctions du runtime.

Chaque langage de programmation a une manière conventionnelle de traduire l'exécution de procédures ou de fonctions, de placer les variables en mémoire et de transmettre des paramètres. Ces conventions sont appliquées par le runtime. Le runtime sert également à mettre en oeuvre

certaines fonctionnalités avancées des langages de programmation telles que le garbage collector qui récupère de l'espace mémoire ² quand c'est nécessaire pour continuer l'exécution du programme.

14.2 Les langages de programmation

14.2.1 Définition de la syntaxe d'un langage de programmation - la BNF

Un langage de programmation est défini formellement grâce à une grammaire formelle, qui inclut des symboles et des règles syntaxiques, auxquels on associe des règles sémantiques. Ces éléments sont plus ou moins complexes selon la capacité du langage.

L'ensemble des syntaxes possibles d'un programme écrit dans un certain langage de programmation (*Python*, *Java* ou *C++* par exemple) (voir <http://docs.python.org/3/reference/> pour *Python3*) est souvent défini grâce au métalangage EBNF (Forme de Backus–Naur Étendu - Extended Backus–Naur Form) qui est un BNF étendu créé par Niklaus Wirth.

Donnons une brève présentation du EBNF utilisé pour définir *Python*.

Ainsi :

```
lc_letter ::= "a"... "z"
name      ::= lc_letter (lc_letter | "_") *
```

définit un élément `lc_letter` comme étant n'importe quelle lettre alphabétique minuscule et `name` comme étant n'importe quelle séquence d'au moins une lettre alphabétique qui peut être suivie de zéro ou plus de caractères `"_"` ou alphabétique.

Dans ce métalangage,

- ce qui est réellement représenté est mis entre " ";
- le symbole `::=` signifie que le nom à gauche *représente* n'importe quel élément compatible avec la définition à droite (par exemple dans `lc_letter ::= "a"... "z"`);
- `"a"... "z"` signifie tous les caractères entre `"a"` et `"z"`;
- `(r1|r2)` (par exemple dans `(lc_letter | "_")`) représente le choix entre la possibilité `r1` et `r2`;
- `r*` (par exemple dans `lc_letter*`) représente la répétition zéro, une ou plusieurs fois un élément de `r`;
- `r+` (par exemple dans `lc_letter+`) représente la répétition une ou plusieurs fois un élément de `r`;
- `[r]` représente l'option: zéro ou une fois `r`.

Ainsi les constantes (*littéraux*) entiers et réel *Python* sont définis (voir http://docs.python.org/3/reference/lexical_analysis.html) par :

² sur le tas d'exécution (runtime heap)

```
integer      ::= decimalinteger | octinteger | hexinteger | bininteger
decimalinteger ::= nonzerodigit digit* | "0"+
nonzerodigit ::= "1"..."9"
digit        ::= "0"..."9"
octinteger   ::= "0" ("o" | "O") octdigit+
hexinteger   ::= "0" ("x" | "X") hexdigit+
bininteger   ::= "0" ("b" | "B") bindigit+
octdigit     ::= "0"..."7"
hexdigit     ::= digit | "a"..."f" | "A"..."F"
bindigit     ::= "0" | "1"
```

et

```
floatnumber  ::= pointfloat | exponentfloat
pointfloat   ::= [intpart] fraction | intpart "."
exponentfloat ::= (intpart | pointfloat) exponent
intpart      ::= digit+
fraction     ::= "." digit+
exponent     ::= ("e" | "E") ["+" | "-"] digit+
```

L’instruction `if` est définie par:

```
if_stmt ::= "if" expression ":" suite
          ( "elif" expression ":" suite ) *
          ["else" ":" suite]
```

14.2.2 Définition de la sémantique d’un langage de programmation

La sémantique définit le sens de chacune des phrases qui peuvent être construites dans le langage, en particulier quels seront les effets de la phrase lors de l’exécution du programme

La sémantique est généralement décomposée en sémantique *statique* et sémantique *dynamique*. La sémantique dépend très fort des paradigmes du langage.

Sémantique statique

Dans un langage impératif, comme *C++*, *Java*, *Python*, l’analyse sémantique statique appelée aussi gestion de contexte s’occupe des relations non locales ; elle s’occupe ainsi :

- du contrôle de visibilité et du lien entre les définitions et utilisations des identificateurs;
- du contrôle de type des “objets”, nombre et type des paramètres de fonctions;
- du contrôle de flot (vérifie par exemple qu’un `goto` est licite).

Pour les langages compilés, il s’agit en gros, de tout ce qui peut être contrôlé pendant la compilation.

Sémantique dynamique

Pour un langage compilé, c'est la partie de contrôle qui a lieu lors de l'exécution du programme. Par exemple, si le type précis d'une variable ne peut être connue qu'à l'exécution, le contrôle de type est dynamique.

Note: Si un programme s'exécute sans erreur (exception,...) mais ne fournit pas le bon résultat, on parle d' *erreur de logique* ou d' *erreur de logique de conception*.

14.2.3 Types

Un type de donnée définit les valeurs que peut prendre une donnée, ainsi que les opérateurs qui peuvent lui être appliqués.

Un type peut être:

- simple (prédéfinis ou non): par exemple: booléen, entier, réel, pointeur (c'est-à-dire adresse), ou
- composé : par exemple, string, tuple, dictionnaire, liste, structure, classe.

Si on prend l'exemple de C++ on y trouve des variables simples (par exemple entière), des pointeurs et des références.

Par exemple:

```
int i = 18;
int *p = &i;
int &j = i;
int tab[5] = {0, 0, 0, 0, 0};
*p = 36;
j = 36;
```

déclare quatre "éléments" que le programme va pouvoir manipuler:

- une **variable** `i` entière, initialisée à la valeur 18;
- une **variable** `p` de type pointeur vers un entier, initialisée à l'adresse de la variable `i`;
- une **constante** `j` de type référence qui est un **alias** (autre nom) à `i`;
- une **constante** qui est un pointeur (une adresse) vers un tableau de 5 éléments entiers initialisés à 0, c'est-à-dire une zone mémoire qui contient 5 variables entières initialisées à 0.

Dans ce programme

- `i` et `j` sont vues comme deux *noms* pour la même variable. On peut donc faire le parallèle avec le code *Python* `j = i = 18`;
- `tab` est similaire au code *Python* `tab = [0]*5` qui définit une liste de 5 éléments nuls;

Par contre, *Python* n'a pas la notion de pointeur, qui permet explicitement de manipuler des adresses dans des variables et qui permet tant d'amusement en programmation C++ (y compris l'absence de garbage collector).

Note: En pratique *pointeur* (comme `p` dans le programme plus haut) et *référence* (comme `j`) manipulent tous les deux des adresses. La différence est que dans le cas du pointeur, si l'utilisateur veut manipuler la variable ou l'objet "pointé", il doit l'exprimer explicitement (comme avec `*p = 36` qui assigne à la variable pointée par `p`, c'est-à-dire `i`, la valeur 36). Avec les références, le *déréférencage* (dereferencing) est automatique; on ne peut manipuler l'adresse mais uniquement l'objet référencé (comme avec `j = 36` qui a le même effet que l'instruction précédente).

Typage statique et typage dynamique

On parle de typage statique (exemple: avec C++, Java, Pascal) quand la majorité des vérifications de type sont effectuées au moment de la compilation. Au contraire, on parle de typage dynamique (exemple avec SmallTalk Common Lisp, Scheme, PHP, Perl, Tcl, Python, Ruby, Prolog) quand ces vérifications sont effectuées pendant l'exécution.

Les avantages des langages statiques par rapport aux dynamiques sont :

- beaucoup d'erreurs sont découvertes plus tôt (grâce au contrôle de type à la compilation);
- le fait de contraindre les programmeurs à bien structurer leur code;
- permet d'avoir un code compilé plus rapide où le contrôle de type est (en grande partie) terminé après la compilation;
- fixe les types ce qui permet un code plus simple à comprendre

Les avantages des langages dynamiques par rapport aux langages statiques sont :

- puissance et flexibilité: permet de faire des choses difficiles à implémenter avec des langages statiques (polymorphisme, introspection, exécution de code dynamique, ...).

Typage fort et typage faible

Un langage de programmation est dit fortement typé lorsqu'il garantit que les types de données employés décrivent correctement les données manipulées. Par opposition, un langage sans typage fort peut être faiblement typé, ou pas du tout typé (mais en pratique ce n'est jamais le cas). BASIC, JavaScript, Perl, PHP sont plutôt à mettre dans la catégorie des langages faiblement typés; C++, C#, Java, Python, OCaml dans les langages fortement typés.

Typage explicite et typage implicite

Avec un typage explicite, c'est à l'utilisateur d'indiquer lui-même les types qu'il utilise, par exemple lors des déclarations de variables ou de fonctions.

Par exemple, en langage C, le typage est explicite :


```
int i = 0; // cette déclaration indique explicitement que
           // la variable i est de type entier
```

Au contraire, avec un système de typage implicite, le développeur laisse au compilateur ou au runtime le soin de déterminer tout seul les types de données utilisées, par exemple par inférence.

Python a donc un typage **dynamique, fort et implicite**.

14.3 Paradigmes des langages de programmation

Chaque langage de programmation reflète un ou plusieurs paradigmes de programmation. Un paradigme est un ensemble de notions qui oriente le travail de réflexion du programmeur et peut être utilisé pour obtenir une solution à un problème de programmation. Chaque paradigme amène une technique différente de programmation; une fois qu'une solution a été imaginée par le programmeur selon un certain paradigme, un langage de programmation qui suit ce paradigme permettra de l'exprimer.

14.3.1 Le paradigme impératif ou procédural

est basé sur l'idée d'une exécution étape par étape semblable à une recette de cuisine. Il est basé sur le principe de la machine de Von Neumann. Un ensemble de structures permet de contrôler l'ordre dans lequel sont exécutées les commandes qui décrivent les étapes. L'abstraction est réalisée à l'aide de procédures auxquelles sont transmises des données. Il existe une procédure principale, qui est la première à être exécutée, et qui peut faire appel à d'autres procédures pour effectuer certaines tâches ou certains calculs. Les langages de programmation C, Pascal, Fortran, COBOL, Java, Python sont en paradigme impératif.

14.3.2 Le paradigme fonctionnel

est basé sur l'idée d'évaluer une formule, et d'utiliser le résultat pour autre chose³. Tous les traitements sont faits en évaluant des expressions et en faisant appel à des fonctions, et l'exécution étape par étape n'est pas possible dans le paradigme fonctionnel. Le résultat d'un calcul sert de matière première pour le calcul suivant, et ainsi de suite, jusqu'à ce que toutes les fonctions aient produit un résultat. ML et Lisp sont des langages de programmation en paradigme fonctionnel. Python possède également des notions permettant de programmer dans ce paradigme.

Note: La récursivité est une technique naturellement associée au paradigme fonctionnel.

³, selon le modèle du lambda-calcul

14.3.3 Le paradigme logique

est basé sur l'idée de répondre à une question par des recherches sur un ensemble, en utilisant des axiomes, des demandes et des règles de déduction. L'exécution d'un programme est une cascade de recherche de données dans un ensemble, en faisant usage de règles de déduction. Les données obtenues, et associées à un autre ensemble de règles peuvent alors être utilisées dans le cadre d'une autre recherche. L'exécution du programme se fait par évaluation, le système effectue une recherche de toutes les affirmations qui, par déduction, correspondent à au moins un élément de l'ensemble. Le programmeur exprime les règles, et le système pilote le processus. Prolog est un langage de programmation en paradigme logique.

14.3.4 Dans le paradigme orienté objet,

chaque objet est une entité active, qui communique avec d'autres objets par échange de messages. Les échanges de message entre les objets simulent une évolution dans le temps d'un phénomène réel. Les procédures agissent sur les données et le tout est cloisonné dans des objets. Les objets sont groupés en classes ; les objets d'une même classe sont similaires. La programmation consiste à décrire les classes. Les classes sont organisées selon une structure hiérarchique où il y a de l'héritage: de nouveaux objets peuvent être créés sur la base d'objets existants. Ce paradigme a été développé pour parer aux limitations de son prédécesseur, le paradigme procédural, tout particulièrement avec les très grands programmes. Le paradigme orienté objet aide le programmeur à créer un modèle organisé du problème à traiter, et permet d'associer fortement les données avec les procédures. Simula, Smalltalk, C++ et Java sont des langages de programmation en paradigme orienté objet. Python possède également des notions permettant de programmer dans ce paradigme.

14.4 Histoire des langages de programmation

Note: Cette section résume succinctement *l'Histoire de l'informatique liée aux langages de programmation*. Les références données en début de chapitre sont intéressantes pour voir plus généralement l'Histoire de l'informatique et en particulier les aspects matériels ou principes mathématiques ou algorithmiques sous-jacents (ordinateurs, théorie de l'information et du codage, théorie des langages, théorie de la décidabilité et de la calculabilité, algorithmique, ...).

14.4.1 La préhistoire (avant 1940)



- 820 : Le mathématicien perse Al Khawarizmi publie à Bagdad un traité intitulé "Abrégé du calcul par la restauration et la comparaison" qui, importé en Europe Occidentale lors des invasions arabes aura une grande influence sur le développement des mathématiques.



- En 1801, le métier Jacquard utilisait des trous dans des cartes perforées pour représenter les mouvements du bras du métier à tisser, et ainsi générer automatiquement des motifs décoratifs.
- 1840 : Collaboratrice de Babbage, Ada Lovelace, mathématicienne, définit le principe des itérations successives dans l'exécution d'une opération. En l'honneur du mathématicien perse Al Khawarizmi (820), elle nomme le processus logique d'exécution d'un programme : algorithme (déformation de Al Khawarizmi). Ada Lovelace a traduit le mémoire du mathématicien italien Luigi Menabrea sur la Machine analytique, la dernière machine proposée par Charles Babbage.
- 1854 : Boole publie un ouvrage dans lequel il démontre que tout processus logique peut être décomposé en une suite d'opérations logiques (ET, OU, NON) appliquées sur deux états (ZERO-UN, OUI-NON, VRAI-FAUX, OUVERT-FERME).
- 1887: Herman Hollerith a créé une machine à cartes perforées qui a servi au recensement de 1890.
- 1863-1895 : On peut également considérer les rouleaux en papier d'un pianola (piano mécanique) comme un programme limité à un domaine très particulier, quoique non destiné à être exploité par des humains.

14.4.2 Les années 1940

- 1943 : Le Plankalkül imaginé par Konrad Zuse (implémenté fin des années nonantes)
- 1943 : Le langage de programmation de l'ENIAC
- 1948 : Le langage machine du premier programme enregistré, i.e. le jeu d'instructions de la SSEM : première machine à programme enregistré.

Note: En 1945, un insecte coincé dans les circuits bloque le fonctionnement du calculateur Mark I. La mathématicienne Grace Murray Hopper décide alors que tout ce qui arrête le bon fonctionnement d'un programme s'appellera BUG.

Il faut noter que le terme BUG était déjà utilisé avant cela : Thomas Edison par exemple avait employé ce terme dans un courrier où il parlait de la mise au point problématique de l'une de ses inventions.

14.4.3 Les années 1950 et 1960

- 1950 : Invention de l'assembleur par Maurice V. Wilkes de l'université de Cambridge. Avant, la programmation s'effectuait directement en binaire.
- 1951 : Invention du premier compilateur A0 par Grace Murray Hopper qui permet de générer un programme binaire à partir d'un code source.



Les trois premiers langages de programmation modernes ont été conçus :

- 1957 : FORTRAN (langage impératif), le traducteur de formules (FORmula TRANslator), inventé par John Backus et al. (voir exemple)



- 1958 : LISP (langage impératif et fonctionnel) , spécialisé dans le traitement des listes (LISt Processor), inventé par John McCarthy et al. (exemple: fonction factorielle)



- 1960 : COBOL (langage impératif), spécialisé dans la programmation d'application de gestion (COmmon BUsiness ORiented Language), créé par le Short Range Committee dans lequel on retrouve entre autres Grace Hopper. (voir exemple)

Les descendants de ces trois langages sont actuellement encore très utilisés.

Une autre étape clé de ces années a été la publication à Zurich par une commission d'informaticiens européens et américains d'un nouveau langage permettant de décrire les problèmes de manière algorithmique : ALGOL (ALGorithmic ORiented Language). Le rapport, publié pour la première fois en 1958, fait la synthèse des principales idées circulant à l'époque, et propose deux innovations majeures :


- Structure en blocs imbriqués : le programme peut être structuré en morceaux de codes logiques sans avoir besoin de donner un nom explicite à ce bloc de code ; on pourrait rapprocher cette notion du paragraphe dans le monde littéraire.
- Notion de portée : un bloc peut manipuler des variables qui lui sont propres ; aucun code en dehors de ce bloc ne peut y accéder, et encore moins les manipuler.

La manière même dont le langage a été décrit est innovante en soi : la syntaxe du langage a été décrite de manière mathématique, en utilisant le métalangage BNF (Forme de Backus–Naur). Presque tous les langages à venir utiliseront une variante de cette notation BNF pour décrire leur syntaxe (ou au moins la sous-partie non-contextuelle de leur syntaxe).

Les idées essentielles d'Algol se retrouvent finalement dans Algol 68 :

- La syntaxe et la sémantique deviennent encore plus orthogonales, avec des procédures anonymes, un système de types récurifs, des fonctions d'ordre supérieur...

Niklaus Wirth abandonne alors la commission de conception d'Algol, et à partir de ses travaux

sur Algol-W (pour Wirth) créera un langage plus simple, le Pascal.  Vue d'ensemble :

- 1951 - Regional Assembly Language
- 1952 - AUTOCODE
- 1955 - FLOW-MATIC, ou encore B-0 (Business Language version 0) [ancêtre de COBOL]
- 1957 - FORTRAN

- 1957 - COMTRAN (COMmercial TRANslator) [ancêtre de COBOL]
- 1958 - LISP
- 1958 - ALGOL 58
- 1959 - FACT (Fully Automated Compiling Technique) [ancêtre de COBOL]
- 1960 - COBOL
- 1962 - APL (A Programming Language)
- 1962 - Simula I (Simple universal language)
- 1964 - BASIC (Beginner's All-purpose Symbolic Instruction Code)
- 1964 - PL/I (Programming Language number 1) développé par IBM

14.4.4 1967 à 1978 : mise en place des paradigmes fondamentaux

Véritable foisonnement des langages de programmation. La plupart des paradigmes des principaux langages sont inventés durant cette période :

- Simula 67, inventé par Nygaard et Dahl comme sur-couche d'Algol 60, est le premier langage conçu pour pouvoir intégrer la programmation orientée objet.
- C, un des premiers langages de programmation système, est développé par Dennis Ritchie et Ken Thompson pour les laboratoires Bell entre 1969 et 1973.
- Smalltalk (milieu des années 1970) est l'un des premiers langages de programmation à disposer d'un environnement de développement intégré complètement graphique.
- Prolog (PROgrammation LOGique), défini en 1972 par Colmerauer, Roussel et Kowalski est le premier langage de programmation logique.
- ML (Meta Language) inventé par Robin Milner en 1973, construit sur un typage statique fort et polymorphe au-dessus de Lisp, pionnier du langage de programmation généraliste fonctionnel.

Chacun de ces langages a donné naissance à toute une famille de descendants, et la plupart des langues modernes comptent au moins l'un d'entre eux dans son ascendance.

Note: Les années 1960 et 1970 ont également été l'époque d'un considérable débat sur le bien-fondé de la programmation structurée, qui signifie essentiellement la programmation sans l'utilisation de GOTO (ou break). Ce débat a été étroitement lié à la conception de langages : certaines langages n'intégrant pas GOTO, beaucoup sont donc contraints d'utiliser la programmation structurée. Bien que ce débat eût fait rage à l'époque, désormais un très large consensus existe parmi les programmeurs : même dans des langages qui intègrent GOTO, utiliser cette instruction est devenu quasiment tabou, sauf dans de rares circonstances.

Voici une liste de quelques langages importants qui ont été développés au cours de cette période:

- 1967 - Simula

- 1970 - Pascal
- 1970 - Forth
- 1972 - C
- 1972 - Smalltalk
- 1972 - Prolog
- 1973 - ML
- 1978 - SQL (Structured Query Language), au départ seulement un langage de requêtes, puis étendu par la suite à la construction de programme procédural de type SQL/PSM, PL/SQL,...

14.4.5 Les années 1980 : consolidation, modules, performance

Années d'une relative consolidation.

- C++ combine la programmation système et orientée-objet.
- Le gouvernement des États-Unis normalise Ada, un langage de programmation système destiné à être utilisé par les sous-traitants de la défense.
- Au Japon et ailleurs, des sommes énormes ont été dépensées pour étudier ce qu'on appelle les langages de « cinquième génération » des langages qui intègrent la logique de construction des programmes.
- Les groupes de langages fonctionnels continuent à normaliser ML et Lisp ces idées élaborées et inventées pendant la décennie précédente plutôt que d'inventer de nouveaux modèles.

Tendances:

- utilisation de modules;
- adaptation à de nouveaux contextes (par exemple, les langages systèmes Argus et Emerald adaptés à la programmation orientée-objet pour les systèmes distribués);
- progrès dans la mise en oeuvre des langages de programmation (pour architectures RISC, améliorations de plus en plus agressive des techniques de compilation)

Voici une liste de quelques langages importants qui ont été développés au cours de cette période:

- 1983 - Ada
- 1983 - C++
- 1983 - LaTeX (édition de textes scientifiques)
- 1985 - Eiffel
- 1987 - Perl

14.4.6 Les années 1990

voient peu de nouveautés fondamentales apparaître. Au contraire, beaucoup d'idées anciennes sont combinées et portées à maturité, dans l'objectif principal d'augmenter la productivité du programmeur. Beaucoup de langages à « développement rapide d'application » (RAD : rapid application development) apparaissent. Ce sont souvent des descendants de langages plus anciens. Ils sont généralement orientés objet et fournis avec un environnement de développement intégré ainsi qu'avec un ramasse-miettes (garbage collector). C'est ainsi que sont apparus Object Pascal, Visual Basic et C#. Java est un langage orienté-objet qui implémente également le ramasse-miettes, et a été un important centre d'intérêt.

Les nouveaux langages de script ont une approche plus novatrice et aussi plus radicale. Ceux-ci ne descendent pas directement d'autres langages : ils présentent de nouvelles syntaxes et de nouvelles fonctionnalités leur sont incorporées de manière plus souple. Beaucoup considèrent que ces langages de script sont plus productifs que les langages RAD. Cependant, si les programmes les plus petits sont effectivement plus simples, on peut considérer que des programmes plus gros seront plus difficiles à mettre en oeuvre et à maintenir. En dépit de ces raisons, les langages de script sont devenus les plus utilisés dans un contexte Web.

Voici une liste de quelques langages importants qui ont été développés au cours de cette période:

- 1990 - Haskell
- 1991 - Python
- 1991 - Visual Basic
- 1991 - HTML (Mark-up Language)
- 1993 - Ruby
- 1993 - Lua
- 1995 - Java
- 1995 - Delphi (Object Pascal)
- 1995 - JavaScript
- 1995 - PHP

14.4.7 Les années 2000

L'évolution des langages de programmation continue, à la fois dans l'industrie et la recherche. Quelques tendances actuelles:

- Ajout de sécurité et de sûreté, notamment dans la vérification du langage (analyse statique de programmes), dans le contrôle du flux d'information (format des entrées/sorties) et dans la protection lors de l'exécution de threads simultanés (threadsafe).
- Nouveaux concepts concernant la modularité : les mixin, la délégation, la programmation orientée aspect.
- Le développement orienté composant.

- La métaprogrammation, et en particulier la réflexivité ou la manipulation de l'arbre syntaxique abstrait.
- Concentration sur la distribution et la mobilité.
- Intégration avec les bases de données, en particulier les bases de données relationnelles et XML.
- Internationalisation avec le support de l'Unicode dans le code source : possibilité d'utiliser des caractères spéciaux (autres que ASCII) dans le code source.
- Utilisation généralisée d'XML, notamment pour les interfaces graphiques (XUL, XAML).
- Langages massivement parallèles

Voici une liste de quelques langages importants qui ont été développés au cours de cette période:

- 2000 - ActionScript
- 2001 - C#
- 2001 - Visual Basic .NET
- 2002 - F#
- 2003 - Groovy
- 2003 - Scala
- 2003 - Factor
- 2007 - Clojure
- 2009 - Go
- 2011 - Dart

14.4.8 Personnalités importantes dans l'histoire des langages de programmation

- John Backus, inventeur de Fortran.
- John McCarthy, inventeur de LISP.
- Alan Cooper, développeur de Visual Basic.
- Ole-Johan Dahl, co-créateur du langage Simula et de la programmation orientée objet.
- Edsger Dijkstra, inventeur de l'algorithme de chemin le plus court qui porte son nom. Il avait une aversion de l'instruction GOTO en programmation et publia un article à ce sujet "Go To Statement Considered Harmful" en 1968. Inventeur du langage Algol.
- James Gosling, développeur de Oak, précurseur de Java.
- Tony (C.A.R.) Hoare, inventeur du QuickSort, de la logique de Hoare, du langage de spécification CSP (qui a inspiré le langage Occam) et de la notion de moniteur.

- Grace Hopper conceptrice du premier compilateur en 1951 (A-0 System) et du langage COBOL.
- Anders Hejlsberg, développeur de Turbo Pascal et C#.
- Joseph-Marie Jacquard, inventeur de la programmation par carte perforée des métiers à tisser.
- Kenneth E. Iverson, développeur de APL.
- Donald Knuth, inventeur de TeX, et de nombreux travaux sur la complexité et l'analyse syntaxique LR(k), la programmation lettrée....
- Alan Kay, précurseur de la programmation orientée objet et à l'origine de Smalltalk.
- Brian Kernighan, co-auteur du premier livre sur le langage C avec Dennis Ritchie, co-auteur des langages AWK et AMPL.
- Leslie Lamport, LaTeX, algorithmes distribués.
- Ada Lovelace, première programmeuse au monde (a donné son prénom au langage ADA)
- Yukihiro Matsumoto, créateur de Ruby.
- Bertrand Meyer, inventeur de Eiffel.
- Robin Milner, inventeur de ML, de LCF (le premier système de démonstration automatique de théorèmes) et le langage de spécification de processus concurrents CCS (calculus of communicating systems) et son successeur, le pi-calcul.
- John von Neumann, inventeur du concept de système d'exploitation.
- Kristen Nygaard, co-créateur du langage Simula et de la programmation orientée objet.
- Martin Odersky, créateur de Scala.
- Dennis Ritchie, inventeur du langage C.
- Guido van Rossum, créateur de Python.
- Bjarne Stroustrup, développeur de C++.
- Ken Thompson, inventeur de Unix.
- Larry Wall, créateur de Perl et Perl 6
- Niklaus Wirth inventeur de Pascal et Modula.

RÈGLES DE BONNES PRATIQUES POUR ENCODER EN PYTHON

15.1 PEP

Les recommandations et améliorations pour le langage **Python** sont répertoriées dans des documents appelés PEPs (Python Enhancement Proposals) sur le site officiel <https://www.python.org>. Chaque PEP porte un numéro. Le PEP 20 résume les 20 aphorismes (dont seulement 19 ont été rédigés) utilisés par Guido van Rossum, le BDFL (Benevolent Dictator for Life) Python pour concevoir le langage lui-même.

Voici son contenu

15.1.1 PEP 20: The Zen of Python

- 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!

Note: Lors de vos sessions de codage Python, en cas de doute, vous pouvez faire

```
>>> import this
```

qui vous rappellera le contenu du PEP 20.

Le PEP 8 donne plus concrètement le `style` conseillé pour coder en Python, c'est-à-dire, l'ensemble des conventions d'écriture et de façon de structurer le code, non obligatoires pour qu'un programme soit correct et efficace mais permettant d'en optimiser sa lisibilité et sa compréhension. PEP 8 recommande d'utiliser un style de façon `consistante` c'est-à-dire de garder le même style tout au long du code, sauf quand le respect de ce style nuit à la lisibilité.

Warning: Règle de bonne pratique
Suivre les recommandations faites dans PEP 8

Dans ce qui suit nous reprenons les recommandations qui nous semble les plus importantes.

15.2 Convention de codage

15.2.1 Indentation

La règle simple est d'ajouter 4 caractères pour chaque nouvelle indentation. Ne pas utiliser de tabulation.

15.2.2 Lignes de continuation

Les lignes doivent avoir moins de 79 caractères (72 pour les **Docstrings**). Les blocs de texte plus longs s'écrivent sur plusieurs lignes. Il se peut que le caractère de continuation backslash ('`\`') soit nécessaire, même si la plupart du temps il peut être évité.

Note: Exemple où le backslash est nécessaire pour continuer le bloc sur plusieurs lignes

```
x = "ceci est un long texte" + \  
    "qui prend plus d'une ligne"
```

Warning: Bonnes pratiques

Par ordre de préférence:

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                        var_three, var_four)

# More indentation included to distinguish this from the rest.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)

# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

Pour les listes:

Warning: Bonnes pratiques

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

15.2.3 Lignes blanches

On peut de façon parcimonieuse, mettre une ligne blanche pour séparer des groupes de fonctions qui ne sont pas logiquement liées ou pour mettre en évidence une nouvelle partie dans une fonction.

15.2.4 Espaces

Pas d'espace juste après une parenthèse ouvrante ou avant une parenthèse fermante. espace après une virgule.

Warning: Bonnes pratiques

```
spam(ham[1], {eggs: 2})

if x == 4:
    print(x, y)
    x, y = y, x
spam(1)
dict['key'] = list[index]
x = 1
y = 2
long_variable = 3
```

15.3 Règles de bonnes pratiques pour concevoir un programme

Nous avons rédigé un document qui tient sur une page, et disponible ici, reprenant un ensemble de recommandations que nous voulons donner au programmeur débutant. Le texte de ce document est repris ci-dessous.

15.3.1 Traduction d'un problème en programme

Analysez le problème

- Identifiez clairement ce que sont les *données fournies*, ainsi que les *résultats et types* attendus à l'issue du traitement.
- Formalisez une *démarche générale de résolution* par une séquence d'opérations simples.
- Vérifiez que vous *envisagez tous les cas* de figures (en particuliers les cas *limites*).

Découpez votre problème en fonctions

- Chaque fonction doit réaliser **une** tâche clairement identifiée.
- Limitez les fonctions à *15 lignes* maximum, sauf dans des cas exceptionnels.
- *Eviter la redondance* dans le code (copier/coller). Si cela arrive, c'est qu'il manque soit une fonction, soit une boucle, soit que des tests conditionnels peuvent être regroupés.
- N'utilisez *pas de variables globales*.
- Veillez à ce que tous les paramètres et variables d'une fonction soient *utilisés* dans cette fonction.
- Pour une fonction qui renvoie un résultat, organisez le code pour qu'il ne contienne qu'un seul *return*, placé comme dernière instruction de la fonction.

- Ne modifiez pas les *paramètres*. Exemple: si vous recevez une borne inférieure *first* et une supérieure *last* et que vous devez itérer de la première à la dernière, n'incrémentez pas *first* dans la boucle, car la signification n'en serait plus claire; créez plutôt une variable locale *pos* initialisée à *first*.
- Sauf si la fonction a comme but de modifier la (structure de) données reçue en paramètre; dans ce cas la fonction ne renvoie pas de valeur.

Testez le code au fur et à mesure du développement

- Créez des *scénarios de test*, pour lesquels vous choisissez les données fournies et vous vérifiez que le résultat de la fonction est conforme à ce que vous attendez.
- Vérifiez les cas particuliers et les *conditions aux limites*. Exemple: pour le calcul d'une racine carrée, que se passe-t-il lorsque le paramètre est un nombre négatif?

15.3.2 Programmation

Style de programmation

- Utilisez la forme raccourcie *if(is_leap_year(2008))* plutôt que la forme équivalente *if(is_leap_year(2008) == true)*
- Utilisez la forme *return expression_bool* plutôt que la forme équivalente

```
if expression bool :  
    res = true  
else:  
    res = false  
return res
```

- N'exécutez pas plusieurs fois une fonction alors qu'une exécution suffit en retenant le résultat.
- Précisez le domaine de validité des paramètres et gérez les erreurs éventuelles avec des exceptions.
- N'utiliser pas les exceptions pour faire des tests liés à l'algorithme et non à la gestion des erreurs.

Quelques erreurs classiques

- Vous essayez d'utiliser une variable avant de l'avoir *initialisée*.
- L'*alignement des blocs* de code n'est pas respecté.
- Vous oubliez de fermer un fichier que vous avez ouvert.

15.3.3 Nommage de variables, fonctions, etc.

Utilisez une convention de nommage

- *joined_lower* pour les *variables* (attributs), et *fonctions* (méthodes)
- *ALL_CAPS* pour les *constantes*
- *StudlyCaps* pour les *classes*

Choisissez bien les noms

- Donner des noms de variables qui expriment leur contenu, des noms de fonctions qui expriment ce qu'elles font (cf. règles de nommage ci-dessus).
- Codez si possible en *anglais*.
- Évitez les noms trop proches les uns des autres.
- Utilisez aussi systématiquement que possible les mêmes genres de noms de variables. Exemples: *i, j, k* pour des indices, *x, y, z* pour les coordonnées, *max_length* pour une variable, *is_even()* pour une fonction, etc.

15.3.4 Documentation du code

Soignez la clarté de votre code

... c'est la première source de documentation.

- Utilisez les *docstrings* dans chaque fonction pour : (1) brièvement décrire ce que fait la fonction, **pas comment elle le fait**, et préciser ses entrées et sorties, (2) décrire les arguments des fonctions.
- Soignez les indentations (2 à 4 espaces chacune) et à la gestion des espaces et des lignes blanches.
- Il faut commenter le code à bon escient et avec parcimonie. Évitez d'indiquer le *fonctionnement* du code dans les commentaires. Exemples: Avant l'instruction *for car in line:*, ne pas indiquer qu'on va boucler sur tous les caractères de la *line*....
- Évitez de paraphraser le code. N'utilisez les commentaires que lorsque la fonction d'un bout de code est difficile à comprendre.

15.3.5 Derniers conseils

N'hésitez pas à consulter les manuels sur internet (et autres sources de documentation) pour résoudre vos problèmes concrets.

GLOSSAIRE

__main__ le module `__main__` est le module par défaut

algorithme séquence d'étapes précises et non ambiguës pouvant être exécutées de façon automatique

appel de fonction instruction qui exécute une fonction. Elle consiste en le nom de la fonction suivi par une liste d'arguments entre parenthèses.

argument valeur fournie à une fonction quand une fonction est appelée. Cette valeur est assignée au paramètre correspondant dans le corps fonction.

assignation instruction qui assigne une valeur à une variable (variable = valeur)

boucle infinie une boucle dont la condition de fin n'est jamais satisfaite.

bug une erreur dans un programme

chaîne vide chaîne sans caractère et de longueur 0, représentée par deux apostrophes.

chemin chaîne de caractères qui identifie un fichier.

chemin absolu chemin qui commence au répertoire du plus haut niveau du système.

chemin relatif chemin qui commence depuis le répertoire courant.

code source un programme écrit en langage de haut niveau avant sa compilation ou son interprétation

commentaire information dans un programme destinée au lecteur du code source et qui n'a pas d'effet sur l'exécution du programme

compiler traduire un programme écrit dans un langage de haut niveau en un langage de bas niveau en une fois, en vue de sa future exécution

concaténer joindre bout à bout des chaînes de caractères

condition expression booléenne dans une instruction conditionnelle qui détermine quelle branche doit être exécutée

conditions chaînées instruction conditionnelle avec une série de branches alternatives

conditions imbriquées instruction conditionnelle qui apparaît à l'intérieur d'une autre instruction conditionnelle

constante valeur qui ne peut être modifiée, par opposition aux variables

corps d'une fonction (body) séquence d'instructions dans une définition de fonction.

deboguer le processus d'identification et de correction des 3 types d'erreurs de programmation

décrémenter mettre à jour une variable en diminuant sa valeur (souvent de -1).

définition de fonction instruction qui crée une nouvelle fonction, spécifie son nom, ses paramètres, et les instructions qu'elle doit exécuter.

diagramme d'état représentation graphique d'un ensemble de variables et des valeurs qu'elles réfèrent

dictionnaire séquence mutable de paires "clef-valeur".

division entière opération qui divise deux nombres entiers et retourne un entier (uniquement la partie entière du résultat)

docstring commentaire multiligne placé au début du corps d'une fonction, destiné à sa documentation.

en-tête d'une fonction (header) la première ligne de la définition d'une fonction (def, nom de la fonction, liste d'arguments, caractère "deux points").

erreur de syntaxe une violation des règles d'écriture dans un programme qui le rend impossible à interpréter ou à compiler

erreur sémantique une erreur dans un programme qui fait que quelque chose de différent de ce que le programmeur voulait réaliser se produit

exception une erreur détectée pendant l'exécution du programme

exécutable un programme après compilation, prêt à être exécuté

expression combinaison de variables, d'opérateurs et de valeurs et dont le résultat est une valeur

expression booléenne expression qui retourne soit vrai (True), soit faux (False)

élément (item) une des valeurs d'une séquence.

évaluer simplifier une expression en appliquant les opérations dans le but d'obtenir la valeur résultat

fichier texte séquence de caractères stockée dans un support permanent (disque dur, CD-ROM, etc.).

flot d'exécution ordre dans lequel les instructions sont exécutées dans un programme.

fonction séquence d'instructions qui possède un nom. Les fonctions peuvent prendre des arguments (ou pas) et peuvent retourner une valeur (ou pas : retourne None).

hachable propriété d'un type immuable lui permettant d'être converti de manière déterministe en un nombre entier.

immuable propriété d'une séquence dont les éléments ne peuvent être assignés.

import instruction qui lit un module et crée un objet module.

incrémenter mettre à jour une variable en augmentant sa valeur (souvent de 1).

instruction morceau du code qui représente une commande ou une action

instruction conditionnelle instruction qui contrôle le flot d'exécution en fonction de certaines conditions

interpréter exécuter un programme écrit dans un langage de haut niveau en le traduisant pas à pas

introspection capacité d'un programme à examiner son propre état

introspection particularité du langage Python qui lui permet de s'auto-explorer.

itération (ou boucle) exécution répétée d'un ensemble d'instructions.

lancer (une exception) on lance une exception quand quelque chose d'exceptionnel se produit, via l'instruction `raise`.

langage de bas niveau un langage exécutable par un ordinateur (aussi appelé langage machine ou assembleur)

langage de haut niveau un langage comme *Python* facile à lire et écrire pour l'humain

langage impératif langage (de programmation) où un programme est formé explicitement d'instructions que l'ordinateur doit exécuter étape par étape. Il est basé sur le principe de la machine de Von Neumann.

littéral constante dont la représentation est donnée explicitement (exemple: `3`, `3.14`, `'bonjour'`)

méthode fonction qui est associée à un objet et qui est invoquée en utilisant la notation point.

mode interactif une manière d'utiliser Python en tapant les instructions via le prompt

mode script une manière d'utiliser Python en lisant les instructions depuis un fichier

module fichier qui contient une collections de fonctions et d'autres définitions.

mot-clé mot réservé et utilisé par le langage Python pour parser un programme; on ne peut pas utiliser les mots-clef comme nom de variables

notation point (dot notation) syntaxe pour appeler une fonction ou utiliser une variable définie dans un module en spécifiant le nom du module suivi d'un point, et du nom de la fonction ou de la variable. Permet également d'accéder aux attributs d'un objet fonction ou d'un objet module.

objet quelque chose qu'une variable peut référer. Pour le moment, un objet est une *valeur* qui possède des attributs et des méthodes.

objet fonction valeur créée par la définition d'une fonction. Le nom de la fonction est une variable qui réfère à un objet fonction.

objet module valeur crée par une instruction `import` et qui fournit un accès aux valeurs et fonctions définies dans le module.

opérande une des valeurs sur lesquelles un opérateur s'applique

opérateur symbole spécial qui représente une opération simple comme l'addition, la multiplication ou la concaténation de chaînes de caractères

opérateur de comparaison un des opérateurs qui comparent ses opérandes : `==`, `!=`, `<`, `>`, `<=` et `>=`.

opérateur logique un des opérateurs qui combinent des expressions booléennes : `and`, `or` et `not`.

paramètre variable utilisée à l'intérieur d'une fonction qui réfère à la valeur passée en argument.

parser examiner un programme et analyser sa structure syntaxique

portabilité le fait qu'un programme puisse être exécuté sur plus d'une sorte d'ordinateurs

portée (scope) la portée d'une variable est la zone du programme dans laquelle elle est disponible. La portée d'une variable locale ou d'un paramètre est limitée à sa fonction.

postcondition condition qu'un code ou une fonction garantit si l'ensemble des préconditions sont respectées (= la validité de son résultat)

précondition condition que le code ou la fonction suppose vraie pour les entrées, avant d'exécuter son travail

print instruction qui ordonne à l'interpréteur Python d'afficher une valeur à l'écran

programme séquence d'instructions écrites dans un langage de programmation particulier et qui spécifie comment réaliser un calcul ou une tâche

prompt les caractères `>>>` qui indiquent que l'interpréteur est prêt à recevoir des instructions

ratrapper (une exception) on rattrappe une exception dans une clause `except`, pour y exécuter du code spécifique.

règles de précedence ensemble de règles définissant l'ordre dans lequel les expressions impliquant plusieurs opérateurs et opérandes sont évaluées

répertoire (dossier) un répertoire possède son propre nom et contient une collection de fichiers ou d'autres répertoires.

résolution de problème processus de formulation d'un problème (spécification), trouver une solution et exprimer la solution

script un programme stocké dans un fichier (en vue d'être interprété)

sémantique la signification (la logique, le sens) d'un programme

séquence ensemble ordonné, c'est-à-dire un ensemble de valeurs où chaque valeur est identifiée par un index entier.

syntaxe la structure et les règles d'un langage de programmation

traceback liste de fonctions qui sont exécutées, affichées quand une exception se produit.

tranche (slice) partie d'une chaîne spécifiée par un intervalle d'indices `s`.

type classe de valeurs (exemple: entiers (`int`), réels (`float`), chaînes de caractères (`str`))

valeur unité élémentaire de données, comme un nombre ou une chaîne de caractères, qu'un programme manipule

valeur de retour le résultat d'une fonction. Si un appel de fonction est utilisé comme une expression, sa valeur de retour est la valeur de l'expression.

variable nom qui réfère à une valeur

variable locale variable définie à l'intérieur d'une fonction. Une variable locale ne peut être utilisée qu'à l'intérieur de sa fonction.

AIDE-MÉMOIRE *PYTHON* 3.2 - (VERSION OCTOBRE 2012)



17.1 Fonctions

- `int(x)` : convertit `x`, de type `float` ou `str`, en entier
- `float(x)` : convertit `x`, `int` ou `str`, en réel
- `str(x)` : convertit `x`, `int` ou `float`, en `str`
- `list(x)` : convertit `x` en `list`
- `tuple(x)` : convertit `x` en `tuple`
- `help(x)` : aide sur `x`
- `dir(x)` : liste des attributs de `x`
- `type(x)` : type de `x`
- `print ...` : imprime
- `input(x)` : imprime le string `x` et lit le string qui est introduit au clavier
- `round(x)` : valeur arrondie du `float x`
- `len(s)` : longueur de la séquence `s`
- `range([start], stop, [step])` : retourne une suite arithmétique d'entiers

17.2 Modules

- `math` : accès aux constantes et fonctions mathématique (`pi`, `sin()`, `sqrt(x)`, `exp(x)`, `floor(x)` (valeur plancher), `ceil(x)` (valeur plafond), ...) : exemple: `math.ceil(x)`

- copy: copy(s), deepcopy(s): *shallow* et *deepcopy* de s
- pickle:
 - dumps(v) : transforme v en une représentation,
 - loads(r) : reconstitue l'objet
 - dump(v,f) : transforme et écrit dans le fichier f
 - load(f) : reconstitue à partir de la représentation lue de f
- shelve
 - db = open() : créer un fichier comme objet de type shelve
 - db.close() : fermeture

17.3 Opérations et méthodes sur les séquences (str, list, tuples)

- min(s),max(s): élément minimum, maximum
- sum(s): (ne fonctionne pas pour les string): somme de tous les éléments (valeur numérique)
- s.index(value, [start, [stop]]) : premier indice de value dans s[start:stop]
- s.count(sub [,start [,end]]) : le nombre d'occurrences sans chevauchement de sub dans s[start:end]
- map(f,s) : créer une liste où chaque élément i de s est remplacé par f(i)
- filter(f,s) : créer une séquence du même type que s avec les éléments i de s tel que f(i) ou i.f() est vrai
- reduce(f,s) : applique f() deux à deux aux éléments consécutifs de s (de gauche à droite) et renvoie le résultat

17.4 Méthodes sur les str

- s.lower() : string avec caractères en minuscule
- s.upper() : string avec caractères en majuscule
- s.islower(), s.isdigit(), s.isalnum(),s.isalpha(), s.isupper(): vrai si dans s on a (respectivement) des minuscules, des chiffres, des car. alphanumériques, alphabétiques, majuscules
- s.find(sub [,start [,end]]) : premier indice de s où le sous string sub est trouvé dans s[start:end]
- s.replace(old, new[, co]) : retourne une copie de s en remplaçant toutes les (ou les co premières) occurrences de old par new.

- `s.format(...)` : sert au formatage (en particulier) d'output
- `s.capitalize()` : met la première lettre en majuscule
- `s.strip()` : copie de `s` en retirant les *blancs* en début et fin
- `s.join(t)` : créer un str qui est le résultat de la concaténation des éléments de `t` chacun séparé par le str `s`
- `s.split([sep [,maxsplit]])` : renvoie une liste d'éléments séparés dans `s` par le caractère `sep` (par défaut *blanc*); au max `maxsplit` séparations sont faites

17.5 Opérateurs et méthodes sur les listes `s`

- `s.append(v)` : ajoute un élément valant `v` à la fin de la liste
- `s.extend(s2)` : rajoute à `s` tous les éléments de la liste `s2`
- `s.insert(i,v)` : insert l'objet `v` à l'indice `i`
- `s.pop()` : supprime le dernier élément de la liste et retourne l'élément supprimé
- `s.remove(v)` : supprime la première valeur `v` dans `s`
- `s.reverse()` : retourne la liste, le premier et dernier élément échange leurs places, le second et l'avant dernier, et ainsi de suite
- `s.sort(cmp=None, key=None, reverse=False)` : trie `s`
- `del s[i]`, `del s[i:j]` : supprime un ou des éléments de `s`
- `zip(a,b,c)`: construit une liste de des triples dont le *i*ème élément reprend le *i*ème élément de chaque séquence `a,b,c`
- `it=iter(s)` : créé un itérateur
- `it.next()` : élément suivant de l'itérateur s'il existe, exception `StopIteration` sinon

17.6 Méthodes sur les dict

- `d.clear()` : supprime tous les éléments de `d`
- `d.copy()` : *shallow* copie de `d`
- `{ }.fromkeys(s,v)` : crée un dict avec les clés de `s` et valeur `v`
- `d.get(k [,v])` : renvoie la valeur `d[k]` si elle existe `v` sinon
- `d.items()` : liste des items `(k,v)` de `d`
- `d.keys()` : liste des clés
- `d.pop(k [,v])` : enlève `d[k]` et renvoie sa valeur ou `v`
- `d.popitem()` : supprimer un item `(k,v)` et retourne l'item sous forme de tuple

- `d.setdefault(k [,v])` : `d[k]` si elle existe sinon `v` et rajoute `d[k]=v`
- `d.update(s)` : `s` est une liste de tuples que l'on rajoute à `d`
- `d.values()` : liste des valeurs de `d`

17.7 Méthodes sur les fichiers

- `f=open('fichier')` : ouvre 'fichier' en lecture
- `f=open('fichier','w')` : ouvre 'fichier' en écriture
- `f=open('fichier','a')` : ouvre 'fichier' en écriture en rajoutant après les données déjà présentes
- `f.read()` : retourne le contenu du fichier `f`
- `f.readline()` : lit une ligne
- `f.readlines()` : renvoie la liste des lignes de `f`
- `f.write(s)` : écrit la chaîne de caractères `s` dans le fichier `f`
- `f.close()` : ferme `f`

17.8 Exceptions

- ```
try:
 ...
 raise ...
 ...
except:
 ...
else:
 ...
finally:
 ...
```

# INDEX

## Symbols

élément, 206  
évaluer, 206  
\_\_main\_\_, 205

## A

algorithme, 205  
appel de fonction, 205  
argument, 205  
assignation, 205

## B

boucle infinie, 205  
bug, 205

## C

chaîne vide, 205  
chemin, 205  
chemin absolu, 205  
chemin relatif, 205  
code source, 205  
commentaire, 205  
compiler, 205  
concaténer, 205  
condition, 205  
conditions chaînées, 205  
conditions imbriquées, 205  
constante, 206  
corps d'une fonction, 206

## D

décrémenter, 206  
définition de fonction, 206  
deboguer, 206  
diagramme d'état, 206  
dictionnaire, 206  
division entière, 206  
docstring, 206

## E

en-tête d'une fonction, 206  
erreur de syntaxe, 206  
erreur sémantique, 206  
exécutable, 206  
exception, 206  
expression, 206  
expression booléenne, 206

## F

fichier texte, 206  
flot d'exécution, 206  
fonction, 206

## H

hachable, 206

## I

immuable, 206  
import, 207  
incrémenter, 207  
instruction, 207  
instruction conditionnelle, 207  
interpréter, 207  
introspection, 207  
itération, 207

## L

lancer (une exception), 207  
langage de bas niveau, 207  
langage de haut niveau, 207  
langage impératif, 207  
littéral, 207

## M

méthode, 207  
mode interactif, 207  
mode script, 207

module, [207](#)

mot-clé, [207](#)

## N

notation point, [207](#)

## O

objet, [207](#)

objet fonction, [207](#)

objet module, [207](#)

opérande, [207](#)

opérateur, [208](#)

opérateur de comparaison, [208](#)

opérateur logique, [208](#)

## P

paramètre, [208](#)

parser, [208](#)

portée, [208](#)

portabilité, [208](#)

postcondition, [208](#)

précondition, [208](#)

print, [208](#)

programme, [208](#)

prompt, [208](#)

## R

répertoire, [208](#)

résolution de problème, [208](#)

règles de précedence, [208](#)

ratrapper (une exception), [208](#)

## S

sémantique, [208](#)

séquence, [208](#)

script, [208](#)

syntaxe, [208](#)

## T

traceback, [208](#)

tranche, [208](#)

type, [208](#)

## V

valeur, [209](#)

valeur de retour, [209](#)

variable, [209](#)

variable locale, [209](#)

# INFO-F101 : Programmation

## Syllabus d'exercices

Réalisé par l'équipe d'assistants du cours INFO-F101

Département d'Informatique, Faculté des Sciences  
Université libre de Bruxelles

Édition 2014–2015

# Remerciements et historique

**Édition 2013–2014 (réorganisation des exercices, création de nouveaux exercices).** Liran Lerman, Luciano Porretta et Nikita Veshchikov (coordinateur du chantier), Mohamed Amine Youssef.

**Édition 2012–2013 (réorganisation des exercices, création d'exercices préparatoires, passage à Python 3).** Liran Lerman, Markus Lindström (coordinateur du chantier), Luciano Porretta et Nikita Veshchikov. Merci également à Marie Boulvain (conseillère pédagogique du Centre de Didactique Supérieure de l'Académie universitaire Wallonie-Bruxelles) pour l'aide à la rédaction de la préface.

**Édition 2011–2012 (création chapitre « Instructions de base » et compléments sur complexité et invariants).** Stéphane Fernandes Medeiros, Markus Lindström, Naïm Qachri et Alessia Violin.

**Édition 2010–2011 (« Big Bang » : passage de C++ à Python 2).** Stéphane Fernandes Medeiros, Naïm Qachri et Jérôme Vervier. Relecture par Markus Lindström et Alessia Violin. Remerciements également à Hadrien Mélot (UMons) pour nous avoir laissé nous inspirer des exercices de son cours pour réaliser cette édition.

Les exercices du chapitre « Instructions de base » introduit dans l'édition 2011–2012 ont été inspirés par le syllabus du cours de Programmation (anciennement « Algorithmique et Programmation ») des éditions précédentes, orientées vers le langage C++, auquel ont contribué Nadjat Benseba, Emmanuel Dall'Olio, Martin De Wulf, Gilles Geeraerts, Joël Goossens, Christophe Macq, Olivier Markowitch, Thierry Massart et Patrick Meyer. Cet ouvrage a également servi de base principale pour les chapitres « Complexité » et « Logique et invariants ».

# Informations importantes

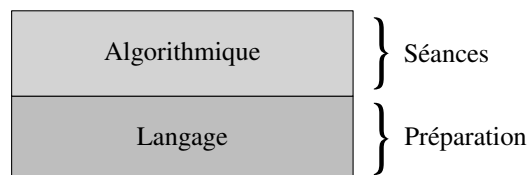
Les séances d'exercices de programmation servent plusieurs objectifs pédagogiques :

- Maîtriser les bases du langage de programmation Python dans des mises en situation ;
- Développer l'esprit logique et la capacité à réaliser des algorithmes ;
- Mettre en pratique quelques bases d'informatique théorique (complexité, logique et invariants) ;
- Inciter l'étudiant à travailler régulièrement.

Chaque chapitre du présent syllabus est séparé en deux parties :

**Préparation.** Chaque chapitre exige l'étude préalable de *prérequis* (typiquement, de la matière vue au cours, ou éventuellement vue en secondaire) ainsi que la résolution éventuelle d'*exercices préparatoires*. Ceux-ci forment une base pour la réalisation des exercices en séance. L'objectif de la préparation est de faire acquérir à l'étudiant les éléments nécessaires de *syntaxe* et de *sémantique* du langage de programmation utilisé.

**Séance(s).** Les exercices en séance sont encadrés par des assistants. Ces derniers constateront si l'étudiant a correctement préparé chaque séance et s'ils peuvent dès lors proposer des problèmes d'*algorithmique* (c'est-à-dire, étant donné les outils fournis par le langage de programmation et leur mise en œuvre, comment peut-on résoudre efficacement un problème donné ?).



Les motivations sous-jacentes à cette découpe sont multiples :

- L'expérience nous a montré que si les étudiants ne maîtrisent pas la syntaxe et la sémantique du langage utilisé, ils ne sont d'une part pas en mesure d'écrire des algorithmes dans ce langage, mais d'autre part ne sont pas non plus armés pour comprendre les explications fournies par les enseignants vu l'absence de vocabulaire commun.
- Les rappels théoriques autrefois donnés en début de TP ne semblaient pas servir d'objectif concret : les étudiants ayant déjà compris la matière perdaient leur temps, tandis que les autres avaient souvent trop de lacunes que pour pouvoir utiliser efficacement l'information fournie. La préparation de la matière avant chaque séance d'exercices devrait pallier ce problème, tout en permettant aux assistants de réduire le temps passé pour des rappels théoriques et d'ainsi pouvoir se concentrer sur la résolution d'exercices et l'acquisition de compétences en algorithmique.

- La préparation individuelle des chapitres confronte l'étudiant à l'interpréteur Python et lui permet de tester son comportement réel, condition *sine qua non* pour la réalisation d'exercices et la réussite du cours.

Les séances sont construites autant que possible pour que les exercices aient une difficulté croissante. Les exercices dénotés par une astérisque \* sont réputés de niveau examen, tandis que ceux dénotés par deux astérisques \*\* sont de difficulté supérieure à ce qui serait demandé à un examen. Ces derniers exercices ont pour but de permettre aux étudiants le souhaitant de se dépasser.

## Règles du jeu

L'étudiant est tenu de préparer chaque chapitre de ce syllabus de manière individuelle en s'aidant du cours théorique. Les exercices préparatoires forment un prérequis nécessaire à la compréhension et à la participation aux séances d'exercices. Si l'étudiant rencontre des difficultés qu'il estime insurmontables dans le cadre des exercices préparatoires, il est invité à contacter l'assistant de son groupe par courrier électronique au moins 24 heures avant la séance (compter un jour ouvrable).

Il est demandé à l'étudiant de respecter le travail de ses condisciples aux séances.

Les corrections d'exercices ne seront pas systématiquement fournies, cela afin d'éviter que l'étudiant vienne aux séances uniquement pour recopier les solutions. L'étudiant aura à sa disposition une plateforme interactive d'apprentissage, nommée *UpyLaB*, lui permettant de tester pour chaque exercice fourni son propre programme et d'avoir une indication sur la validité de ce dernier. La plateforme est accessible à l'adresse suivante :

<http://upylab.ulb.ac.be/>

## Coordonnées des assistants

L'étudiant peut toujours contacter l'assistant de son groupe en dehors des séances d'exercices en fixant un rendez-vous préalable par courriel. Les coordonnées des assistants sont données par chacun d'eux lors de la première séance et sont également disponibles sur l'Université Virtuelle :

<http://uv.ulb.ac.be/>

Des versions électroniques du présent syllabus d'exercices ainsi que des supports de cours et d'anciens examens sont disponibles à cette même adresse.

## Guidances

Des guidances sont à disposition de l'étudiant à partir du mois d'octobre. Toutes les informations relatives à ce service sont disponibles à l'adresse suivante :

<http://www.ulb.ac.be/di/guidances/>



# Table des matières

|           |                                 |           |
|-----------|---------------------------------|-----------|
| <b>1</b>  | <b>Séance introductive</b>      | <b>1</b>  |
| <b>2</b>  | <b>Contrôle de flux</b>         | <b>4</b>  |
| <b>3</b>  | <b>Fonctions</b>                | <b>11</b> |
| <b>4</b>  | <b>Chaînes de caractères</b>    | <b>17</b> |
| <b>5</b>  | <b>Listes</b>                   | <b>22</b> |
| <b>6</b>  | <b>Matrices</b>                 | <b>26</b> |
| <b>7</b>  | <b>Dictionnaires</b>            | <b>30</b> |
| <b>8</b>  | <b>Tris</b>                     | <b>35</b> |
| <b>9</b>  | <b>Complexité</b>               | <b>37</b> |
| <b>10</b> | <b>Logique et invariants</b>    | <b>42</b> |
| <b>11</b> | <b>Récursivité</b>              | <b>48</b> |
| <b>12</b> | <b>Fichiers et exceptions</b>   | <b>50</b> |
| <b>13</b> | <b>Introduction aux classes</b> | <b>54</b> |
| <b>14</b> | <b>Base de données</b>          | <b>57</b> |
| <b>15</b> | <b>Révision</b>                 | <b>61</b> |
| <b>A</b>  | <b>Swampy Turtle</b>            | <b>66</b> |
| <b>B</b>  | <b>Python Imaging Library</b>   | <b>69</b> |

# Chapitre 1

## Séance introductive

### Exercices préparatoires

Matière à réviser :

- les variables ;
- les fonctions `print()` et `type()` ;
- les opérateurs `+`, `-`, `*`, `%`, `/`, `//`, `**`, `=`, `+=`, `-=`, `*=`, `/=`, `//=` ;
- les commandes en terminal `ls` et `cd`.

Lisez le *Mode d'emploi introductif pour les salles du NO4 et NO3* partie 1 « Introduction » et la partie 2 « La console Linux » (seulement 2.1, commandes `ls` et `cd`).

### Exercices en séance

**Ex. 1.1.** Allez dans le menu Applications et lancez l'application *Terminal* située dans l'onglet Accessoires. Lancez *Python 3* en tapant la commande `python3` dans le Terminal.

**Ex. 1.2.** Entrez les expressions suivantes dans l'interpréteur et regardez le résultat :

1. `5`
2. `5 + 1`
3. `x = 5`
4. `x + 1`
5. `x = x + 1`
6. `x`

**Ex. 1.3.** Utilisez l'interpréteur pour afficher les textes suivants (rappel : `print()`) :

1. `Hello World`
2. `Aujourd'hui`
3. `C'est "Domage !"`
4. `Hum \0/`

Allez dans le menu Applications et lancez l'application *pluma* (Éditeur de texte) située dans l'onglet Accessoires (alternativement, tapez « `pluma&` » dans le terminal). Créez un fichier avec les commandes que vous avez tapées pour les exercices précédents dans le terminal. Lancez encore un terminal et exécutez le programme que vous venez d'écrire à l'aide de la commande

`python3 nom_de_fichier.py` (positionnez-vous d'abord dans le bon dossier à l'aide des commandes `cd` et `ls`).

Exemple d'un tel fichier :

```
Mon fichier Python
commande1
commande2
...
commandeN
```

**Ex. 1.4.** Évaluez (à la main) les expressions suivantes et essayez de deviner le type du résultat. Utilisez ensuite l'interpréteur Python pour vérifier vos réponses (rappel : `type()`). Résolvez les exercices avec divisions en utilisant l'opérateur `/`, puis l'opérateur `//`.

1.  $14 - 14$
2.  $1 + 6.9$
3.  $1.0 + 2.0$
4.  $\frac{18}{7+1}$
5.  $\frac{(3+2)*2.5}{4*2}$
6.  $0 * 0.0$
7.  $4^{0.5}$
8.  $\frac{5}{8}$
9.  $-\frac{1}{2}$
10.  $3^{-\frac{1}{2}}$

**Ex. 1.5.** Certaines des lignes de code suivantes contiennent des erreurs. Il peut s'agir d'erreurs syntaxiques ou sémantiques et certaines lignes génèrent des exceptions. Indiquez pour chacune d'entre elles le type d'erreur (s'il y en a) ou le résultat et expliquez brièvement. Vérifiez ensuite à l'aide de l'interpréteur. Le résultat désiré par le programmeur est indiqué en gras.

1. `print (''Bonjour')`  
**Bonjour**
2. `'bla' * 3.0`  
**'blablaba'**
3. `((1 + 4) / (6 * 2))`  
**0.4166666666666667**
4. `int(''14'')`  
**14**
5. `int('3+4')`  
**7**
6. `'3 * 3' * 3 ** 2`  
**81**
7. `3 + 2 / 0 + 2`  
**2**
8. `(print 'Il y a 31 jours en janvier')`  
**Il y a 31 jours en janvier**

```
9. print ("Chuck Norris was born in " + 1940)
 Chuck Norris was born in 1940
```

**Ex. 1.6.** Résolvez les problèmes suivants en écrivant des petits programmes dans des fichiers séparés. Créez d'abord toutes les variables nécessaires, tapez ensuite la formule en une seule ligne et affichez le résultat :

1. Le volume d'une sphère de rayon  $r$  est donné par  $\frac{4}{3}\pi r^3$ . Quel est le volume d'une sphère de rayon 5 ? Et de rayon 8 ?
2. Le prix affiché d'un livre est de 24.95 €, mais vous bénéficiez d'une réduction de 40 %. Par ailleurs, les frais d'envoi sont de 3 €. Quel est le prix total pour 60 livres ? Quel est le prix total de 50 livres si les frais d'envoi sont de 5 € et que vous bénéficiez d'une réduction de 43 % ?
3. Si vous parcourez 10 kilomètres en 43 minutes et 30 secondes, quelle est votre vitesse moyenne en miles par heure ? Quelle est votre vitesse moyenne en miles par heure si vous parcourez 10 kilomètres en 45 minutes ? Pour rappel : 1,61 km = 1 mile, 1 heure = 60 minutes et 1 minute = 60 secondes.
4. L'édition complète de la série « *Les comptes de Chuck Norris* » est composée de 3486 volumes et se trouve dans ses armoires numérotées dans l'ordre. Si chaque armoire peut contenir au plus 89 livres, dans laquelle se trouve le volume numéro 1024 ? Dans quelle armoire se trouverait le volume 404 si chacune pouvait contenir 91 livres ?

**Ex. 1.7.** Supposez que vous ayez quatre variables :  $a$ ,  $b$ ,  $c$  et  $d$ . Comment pourriez-vous vous y prendre pour inverser l'ordre des valeurs qu'elles réfèrent sans utiliser l'assignation multiple comme  $a, b, c, d = d, c, b, a$  ? Par exemple, si à l'initialisation,  $a = 1$ ,  $b = "2"$ ,  $c = 3.0$  et  $d = \text{True}$ , comment obtenir "True 3.0 2 1" à l'écran en entrant `print(a, b, c, d)` ?

**Ex. 1.8.\*\*** Même question que dans l'exercice 1.7 en supposant que les 4 variables sont de type `int`. Vous ne pouvez pas utiliser d'autres variables que  $a$ ,  $b$ ,  $c$  et  $d$ .

**Ex. 1.9.** Allez sur le site [Python.org](https://www.python.org) et cherchez dans la documentation du langage (« language reference ») de l'aide à propos de `print`. Comparez avec le résultat obtenu dans l'interpréteur lorsque vous entrez `help('print')`. Cherchez à vous renseigner sur d'autres aspects de Python vus au cours (`int`, `str`, les opérateurs, ...).

# Chapitre 2

## Contrôle de flux

### Exercices préparatoires

Matière à réviser :

- les variables ;
- l'indentation ;
- les fonctions prédéfinies `input()`, `print()`, `range()` ;
- les types `int`, `float`, `bool` ;
- les opérateurs de comparaison et connecteurs logiques (`>`, `<`, `==`, `!=`, `and`, `or`, etc.) ;
- les instructions de branchement conditionnel `if-else`, `if-elif-else` ;
- les boucles `for` et `while`.

Faites tous les exercices préparatoires sur papier et ensuite vérifiez vos réponses à l'aide de l'interpréteur `python3`.

**Prép. 2.1.** Qu'écrit sur *output* le programme suivant quand on lui fournit en *input* les valeurs 2 et 6 ?

```
a = int(input())
a *= 2
b = int(input())
b += a
print(a)
print(b)
```

**Prép. 2.2.** Qu'écrit sur *output* le programme suivant quand on lui fournit en *input* les valeurs 2, 6 et 4 ?

```
a = int(input())
b = int(input())
a = b
b = int(input())
b += a
print(a,b)
```

**Prép. 2.3.** Qu'écrit sur *output* le programme suivant quand on lui fournit en *input* les valeurs 2 et 6 ?

```
b = int(input())
a = int(input())
a = b+1
```

```
print(a)
a = b+1
print(a)
a += 1
print(a, a+1)
```

**Prép. 2.4.** Qu'écrit sur *output* le programme suivant quand on lui fournit en *input* :

1. les valeurs 2 et 6 ?
2. les valeurs 8 et 3 ?
3. les valeurs 3 et 3 ?

```
a = int(input())
b = int(input())
if a > b:
 print(a)
 a = b
print(a)
```

**Prép. 2.5.** Qu'écrit sur *output* le programme suivant quand on lui fournit en *input* :

1. les valeurs 2 et 6 ?
2. les valeurs 8 et 3 ?
3. les valeurs 3 et 3 ?

```
a = int(input())
b = int(input())
if a > b:
 print(a)
 a = b
print(a)
```

**Prép. 2.6.** Donnez la valeur des variables *a*, *b*, *c*, *arret*, *test1*, *test2* et *test3* après chacune des instructions ci-dessous :

```
a = 2
b = 3
c = 4
test1 = True
test2 = (b>=a) and (c>=b)
test3 = test1 or test2
arret = test3 and (not test2)
a += 1
b -= 1
c -= 2
test1 = True
test2 = (b>=a) and (c>=b)
test3 = test1 or test2
arret = arret or test2
```

**Prép. 2.7.** Qu'écrit sur *output* le programme suivant quand on lui fournit en *input* :

1. les valeurs 1 et 6 ?

2. les valeurs -8 et 2.5 ?

```
a = float(input())
b = float(input())
test = (a >= b)
if test or a < 0:
 print("Oui!")
else:
 print("Non!")
```

**Prép. 2.8.** Que fait cette suite d'instructions ?

```
a = 1
while a <= 5:
 print(a, end=' ')
 a += 1
print(a)
```

**Prép. 2.9.** Que fait cette suite d'instructions ?

```
a = 1
while a < 5:
 a += 1
 print(a, end=' ')
print(a)
```

**Prép. 2.10.** Que fait cette suite d'instructions ?

```
i = 0
for j in range(10):
 i += j
print(i)
```

**Prép. 2.11.** Quelle est la différence entre les quatre instructions suivantes ? Expliquez ce qu'elles font et essayez de les utiliser avec une boucle **for**.

```
range(42)
range(42, 69)
range(42, 69, 3)
range(42, -69, -1)
```

## Exercices en séance

**Ex. 2.1.** Écrire un programme qui affiche  $a^{17}$  ( $a$  lu sur *input*) en employant le moins de multiplications possibles, sans utiliser l'opérateur **\*\*** ni de boucle.

**Ex. 2.2.** Écrire un programme qui affiche  $18 \times a$  ( $a$  lu sur *input*) en employant le moins d'instructions possibles, sans utiliser l'opérateur **\*** ni de boucle.

## Branchements conditionnels

**Ex. 2.3.** Écrire un programme qui lit 3 nombres, et qui, si au moins deux d'entre eux ont la même valeur, imprime cette valeur (le programme n'imprime rien dans le cas contraire).

**Ex. 2.4.** Pour chacune des 4 instructions d’affichage (`print`), donner l’ensemble des valeurs de  $a$  pour lesquelles celles-ci seront exécutées.

```
a = int(input())
if a > 0:
 if a > 1:
 if a > 2:
 print(a-2)
 else:
 print(a-1)
 else:
 print(a)
else:
 print("Erreur")
```

**Ex. 2.5.** Écrire le morceau de code qui si,  $a$  est supérieur à 0, teste si  $a$  vaut 1, auquel cas il imprime «  $a$  vaut 1 » et qui, si  $a$  n’est pas supérieur à 0, imprime «  $a$  est inférieur ou égal à 0 ».

**Ex. 2.6.** Écrire le programme qui lit en input trois entiers  $a$ ,  $b$  et  $c$ . Si l’entier  $c$  est égal à 1, alors le programme affiche sur output la valeur de  $a + b$ ; si  $c$  vaut 2 alors le programme affiche la valeur de  $a - b$ ; si  $c$  est égal à 3 alors l’output sera la valeur de  $a \times b$ . Enfin, si la valeur 4 est assignée à  $c$ , alors le programme affiche la valeur de  $a^2 + b \times a$ . Si  $c$  contient une autre valeur, le programme affiche un message d’erreur.

**Ex. 2.7.** Écrire un programme qui imprime la moyenne arithmétique ( $\frac{a+b}{2}$ ) de deux nombres lus sur input.

**Ex. 2.8.** Écrire un programme qui imprime la moyenne géométrique ( $\sqrt{ab}$ ) de deux nombres lus sur input.

**Ex. 2.9.** Écrire un programme qui lit deux nombres  $a$  et  $b$  sur input et qui calcule et affiche le nombre  $c$  tel que  $b$  soit la moyenne arithmétique de  $a$  et  $c$ .

## Boucles simples

**Ex. 2.10.** Écrire l’équivalent des instructions suivantes sans le mot clé `break` :

```
i=0
while i < 10:
 if i > 5:
 break;
 print(i)
 i=i+1
```

**Ex. 2.11.** Écrire l’équivalent des instructions suivantes sans le mot clé `pass` :

```
i=0
while i < 10:
 if i == 5:
 pass
 elif i == 9:
 pass
 pass
 print(i)
 i=i+1
```



**Ex. 2.12.** Écrire l'équivalent des instructions suivantes sans le mot clé `continue` :

```
for i in range(11):
 if i%2 == 0:
 print(str(i)+" est pair")
 continue
 print(str(i)+" est impair")
```

**Ex. 2.13.** Écrire l'équivalent des instructions suivantes avec une boucle `while` :

```
for i in range(a,b+1):
 print(i)
```

**Ex. 2.14.** Remplacez la boucle `while` une boucle `for` dans le code ci-dessous :

```
i = 1
while i < 42:
 print(i)
 i += 1
```

**Ex. 2.15.** Pour cette exercice vous ne pouvez pas utiliser d'instruction `if`. Écrire un programme qui lit un nombre  $n$  sur input et qui affiche tous les nombres naturels :

1. entre 0 et  $n$  compris, de manière croissante (utilisez la boucle `for`);
2. entre 0 et  $n$  compris, de manière décroissante (utilisez la boucle `while`);
3. pairs entre 0 et  $n$  compris, de manière croissante (utilisez la boucle `while`);
4. entre 0 et  $n$ , bornes non comprises, de manière croissante (utilisez la boucle `for`);
5. multiples de 7 entre 0 et  $n$  bornes non comprises, de manière croissante (utilisez la boucle `for`);
6. multiples de 5 entre 0 et  $n$  compris, de manière décroissante (utilisez la boucle `for`);

**Ex. 2.16.** Modifiez le code ci-dessous en utilisant une boucle `for` :

```
i = 1
while i < 42:
 print(i)
 i = 2*i
```

**Ex. 2.17.** Écrire un programme qui calcule la moyenne des résultats d'une interrogation, ces notes étant données en input, la fin des données étant signalée par la valeur sentinelle  $-1$  (on suppose aussi que la suite des notes contient toujours au moins un élément).

Exemple d'exécution :

```
Entrez les notes.
Note 1 : 2
Note 2 : 8
Note 3 : 14
Note 4 : -1
La moyenne est de 8
```

**Ex. 2.18.** Écrivez un mini jeu : le programme choisit un nombre naturel (nombre secret) entre 0 et 100 de manière aléatoire, le joueur doit deviner ce nombre en utilisant le moins d'essais possibles. A chaque tour le joueur peut faire un essais et le programme doit répondre si le nombre

secret est plus grand ou plus petit. Si le joueur trouve le nombre secret, alors le programme doit afficher "Bravo !" ainsi que le nombre d'essais utilisés par le joueur.

Le joueur a maximum 7 essais ; s'il ne devine pas le secret après 7 essais, le programme s'arrête en écrivant "Vous avez perdu !".

Pour générer des nombres aléatoires importez le module `random`. Pour générer un entier aléatoire dans intervalle  $[0; n]$  utilisez la fonction `randint(0, n)`.

Exemple d'exécution :

```
J'ai choisi un nombre secret !
Quel est le secret ? 50
Non, le secret est plus petit !
Quel est le secret ? 25
Non, le secret est plus grand !
Quel est le secret ? 37
Bravo !
Le secret (37) est trouvé en 3 essais.
```

**Ex. 2.19.\*** On peut calculer approximativement le sinus de  $x$  en effectuant la sommation des  $n$  premiers termes de la série (c'est-à-dire la somme infinie)

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

où  $x$  est exprimé en radians.

Réécrivez cette somme sous la forme

$$\sin(x) = \sum_{i=0}^{\infty} f(i, x)$$

On vous demande donc de trouver  $f(i, x)$ . Écrivez ensuite le code calculant de cette manière la valeur de  $\sin(x)$  où  $x$  est lu sur input. Continuez l'addition des termes successifs dans la série jusqu'à ce que la valeur d'un terme devienne inférieur (en valeur absolue) à une constante  $\varepsilon$  (par exemple,  $\varepsilon = 10^{-5}$ , ce qu'on notera `1e-5` en Python).

## Boucles imbriquées

**Ex. 2.20.** Écrire un programme qui lit sur input une valeur naturelle  $n$  et qui affiche à l'écran un carré de  $n$  caractères X de côté, comme suit (pour  $n = 6$ ) :

```
X X X X X X
X X X X X X
X X X X X X
X X X X X X
X X X X X X
X X X X X X
```

**Ex. 2.21.** Variante de l'exercice 2.20, afficher le triangle supérieur droit, comme suit (pour  $n = 6$ ) :

```
X X X X X X
 X X X X X
 X X X X
 X X X
 X X
 X
```

D'autres variantes :

- afficher uniquement le bord du carré ;
- afficher le triangle inférieur gauche (supérieur gauche, inférieur droit) ;
- afficher un rectangle.

**Ex. 2.22.** Refaire l'exercice 2.20 en supposant que  $n$  est impair et en dessinant des  $\circ$  sur les deux diagonales principales à la place des  $x$ . Par exemple, pour  $n = 7$  :

```

○ x x x x x ○
x ○ x x x ○ x
x x ○ x ○ x x
x x x ○ x x x
x x ○ x ○ x x
x ○ x x x ○ x
○ x x x x x ○

```

**Ex. 2.23.** Refaire l'exercice 2.22 sans utiliser de branchement conditionnel (pas de `if-else`).

## Chapitre 3

# Fonctions

### Exercices préparatoires

Pour les exercices sur le module « droite », veuillez revoir la géométrie euclidienne de base, en particulier la manipulation de droites de la forme analytique  $y = ax + b$ .

**Prép. 3.1.** Quel est l'effet des instructions suivantes ?

```
a, b = 1, 3.5
x, y = a, b
a, b = b, a
```

**Prép. 3.2.** Y a-t-il une différence sémantique entre les instructions suivantes ?

```
x, y = a, b
x, y = (a, b)
(x, y) = a, b
(x, y) = (a, b)
```

**Prép. 3.3.** Que vous donnent pour résultat les instructions suivantes ? Expliquez.

```
(1, 2, 3, 4) <= (1, 2, 3, 4, 0)
(1, 2, 3, 4) <= (1, 1, 8, 4, -10)
(1, 2, 3, 4) <= (1)
(1, 2, 3, 4) <= (1,)
```

**Prép. 3.4.** Essayez l'instruction suivante :

```
a, b, c = input('Entrez trois chiffres')
```

avec les données suivantes comme entrées :

```
- 123
- 1 2 3
- abc
- 1b3
- 1
```

**Prép. 3.5.** Que fait le code suivant ?

```
t = tuple(range(10))
for x in t:
 print(x)
```

**Prép. 3.6.** En supposant que  $x$  et  $y$  sont des variables bien définies, la boucle suivante est-elle susceptible de changer leur valeur ? Expliquez.

```
for e in (x,y):
 e.foo()
 e = 4
```

**Prép. 3.7.** Le code suivant est-il correct ?

```
def plus2(x):
 """Renvoie la valeur donnée augmentée de 2."""
 return x+2 # On renvoie x+2
```

**Prép. 3.8.** Le code suivant est-il correct ?

```
def plus2(x):
 """Renvoie la valeur donnée augmentée de 2."""
 return x+2 """On renvoie x+2"""
```

**Prép. 3.9.** Entrez la fonction de l'exercice 3.7 dans l'interpréteur et entrez la commande `help(plus2)`. Expliquez ce qui se passe. Quid si vous tapez `help(bar)` ? Pourquoi y a-t-il une différence en tapant plutôt `help("bar")` ?

**Prép. 3.10.** Qu'affiche le code suivant ? Justifiez votre réponse.

```
def f(x):
 x = 3

x = 4
f(x)
print(x)
```

**Prép. 3.11.** Qu'aurait affiché le code de l'exercice 3.10 si la fonction `f` était plutôt :

```
def f(x):
 x = 3
 return x
```

**Prép. 3.12.** Qu'aurait-il fallu faire dans les exercices 3.10 et 3.11 pour que la variable  $x$  du code principal prenne la valeur 3 après l'appel à cette dernière fonction `f` ?

**Prép. 3.13.** Que fait la fonction suivante ?

```
def f(x,*args):
 print(len(args))
 print("x =", x)
 for arg in args:
 print(arg)
```

Essayez d'appeler cette fonction avec divers paramètres, par exemple `f()`, `f(4)`, `f(1, 2, 3, 4, 5)`.

**Prép. 3.14.** Le code suivant est-il syntaxiquement correct ? Quelle est sa sémantique ?

```
def f(x,y):
 return (x//y)/(x/y)

print(f(y=3,x=4))
```

## Exercices en séance

**Ex. 3.1.** Écrivez une fonction `swap(a, b)` qui échange le contenu de `a` et `b` et renvoie `a` et `b` avec leur nouvelle valeur, afin de pouvoir l'utiliser comme suit : `a, b = swap(a, b)`.

**Ex. 3.2.** Écrivez une fonction `distancePoints()` qui, étant donnés deux points  $(x_1, y_1)$  et  $(x_2, y_2)$  reçus en paramètres sous forme de tuples, calcule et renvoie la distance euclidienne entre ces deux points.

**Ex. 3.3.** Écrivez une fonction `longueur(*points)` qui prend en paramètres un nombre arbitraire de points de coordonnées  $(x, y)$  et calcule la longueur du trait correspondant. Pour calculer la longueur, il faut effectuer la somme de la longueur des segments qui composent le trait. Un segment de droite est composé de deux points consécutifs passés en paramètre. Pour rappel, la distance entre deux points  $(x_1, y_1)$  et  $(x_2, y_2)$  se calcule comme suit :

$$\text{dist} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

**Ex. 3.4.** Écrivez une fonction qui reçoit quatre points du plan en paramètres et calcule et affiche le périmètre du parallélogramme correspondant. Les points  $(x_1, y_1)$ ,  $(x_2, y_2)$ ,  $(x_3, y_3)$  et  $(x_4, y_4)$  correspondent au coin supérieur gauche, au coin supérieur droit, au coin inférieur droit et au coin inférieur gauche. La fonction n'affichera rien et renverra `False` si les côtés ne sont pas égaux deux à deux, elle renverra `True` sinon.

**Ex. 3.5.** Soit l'équation du second degré  $\alpha x^2 + \beta x + \gamma = 0$  où les paramètres  $\alpha$ ,  $\beta$  et  $\gamma$  sont entrés par l'utilisateur. Écrivez une fonction qui en calcule et affiche les solutions (s'il y en a).

**Ex. 3.6.** Dans le module `random`, la fonction `randint(a, b)` renvoie un nombre aléatoire compris entre `a` et `b` (inclus). Écrivez une fonction qui génère 3 nombres aléatoires représentant 3 dés à jouer (à six faces) et qui renvoie `True` si les dés forment un 421, `False` sinon.

**Ex. 3.7.** Considérons les billets et pièces de valeurs suivantes : 20 €, 10 €, 5 €, 2 €, 1 €. Écrivez une fonction `rendreMonnaie` qui prend en paramètres un entier `prix` et un 5-uple  $(x_{20}, x_{10}, x_5, x_2, x_1)$  d'entiers représentant le nombre de billets et de pièces de chaque sorte que donne un client pour payer l'objet dont le prix est mentionné. La fonction doit renvoyer un 5-uple représentant la somme qu'il faut rendre au client, décomposée en billets et pièces (dans le même ordre que précédemment). La décomposition doit être faite en utilisant le plus possible de billets et pièces de grosses valeurs.

**Ex. 3.8.** Écrivez une fonction `duree` qui prend deux paramètres `debut` et `fin`. Ces derniers sont des couples dont la première composante représente une heure et la seconde composante représente les minutes. Cette fonction doit calculer le nombre d'heures et de minutes qu'il faut pour passer de `debut` à `fin`.

Exemple : un appel à `duree((14, 39), (18, 45))` renvoie `(4, 6)`. Notez qu'un appel à `duree((6, 0), (5, 15))` renvoie `(23, 15)` et non `(0, 45)` !

**Ex. 3.9.** Même exercice mais sans utiliser les instructions conditionnelles.

**Ex. 3.10.** Écrivez une fonction `appliquer(a, b, f)` qui prend 2 entiers et une fonction `f(a, b)` en paramètre et qui renvoie le résultat de la fonction `f(a, b)` appliquée aux deux paramètres `a` et `b` si ce résultat est un entier, 0 sinon.

**Ex. 3.11.** Écrivez une fonction `sum(a, b)` qui prend 2 valeurs et renvoie la somme de  $a$  et de  $b$ . Par défaut, la valeur de  $a$  est 0 et la valeur de  $b$  est 1.

**Ex. 3.12.** Écrivez une fonction `sum(a, b)` en lambda calcul qui prend 2 entiers et renvoie la somme de  $a$  et de  $b$ .

**Ex. 3.13.** Écrivez une fonction `sum(a, b)` en lambda calcul qui prend 2 valeurs et renvoie la somme de  $a$  et de  $b$  si ces deux valeurs sont des entiers. Dans le cas contraire, la fonction renvoie 0. Par défaut, la valeur de  $a$  est 0 et la valeur de  $b$  est 1.

**Ex. 3.14.** Écrivez une fonction `nombreImpair()` qui utilise le mot clé `yield` afin d'afficher autant de nombres impairs que l'utilisateur y fait appel (grâce à la fonction `next`).

**Ex. 3.15.** Écrivez un generateur `nombresPair(n)` ( $n$  est le nombre de nombres pairs qu'on veut générer) qu'on peut utiliser avec une boucle `for`. Par exemple :

```
for p in nombresPair(3):
 print(p)
```

**Ex. 3.16.\*** Le  $n$ -ième nombre de Bell, noté  $B_n$ , est le nombre de partitions possibles d'un ensemble de  $n$  éléments. Il se calcule à partir de la formule suivante :

$$B_n = \sum_{k=1}^n \frac{1}{k!} \sum_{i=1}^k (-1)^{k-i} \frac{k!}{i!(k-i)!} i^n$$

On vous demande d'écrire la fonction `bell(n)` qui renvoie le  $n$ -ième nombre de Bell. Vous ne pouvez pas faire appel aux fonctions du module `math`. Nous demandons également que le code soit le plus efficace possible.

## Module « droite »

Pour cet série d'exercices, nous allons créer un ensemble de fonctions permettant de manipuler des droites (non verticales) dans le plan euclidien. Nous allons placer ces fonctions dans un *module*. Veuillez créer un script (fichier) nommé `droite.py` et placez-y l'implémentation des fonctions décrites ci-dessous. Pour rappel, vos fonctions peuvent se servir des autres fonctions déjà implémentées.

**Notations :** Nous représentons une droite  $d$  dans le plan euclidien  $\mathbb{R}^2$  par le tuple  $(a, b) \in \mathbb{R}^2$  de telle manière à ce que :

$$d \equiv y = ax + b$$

De plus, nous représentons un point du plan euclidien  $\mathbb{R}^2$  par un couple de réels.

Implémentez les fonctions suivantes en supposant que toutes les droites traitées sont telles que  $a \neq 0$  (droites ni horizontales, ni verticales) :

**Ex. 3.17.** `droite(p1, p2)`

**Entrée :** Deux points distincts  $p1$  et  $p2$ .

**Sortie :** Un tuple  $(a, b)$  représentant la droite passant par  $p1$  et  $p2$ , `None` si les deux points ont la même abscisse.

**Rappel :**  $a = \frac{y_2 - y_1}{x_2 - x_1}$  et  $b = y_1 - ax_1 = y_2 - ax_2$ .

**Ex. 3.18.** appartient(d, p)

**Entrée :** Une droite d et un point p.

**Sortie :** True si  $p \in d$ , False sinon.

**Rappel :** vérifier si  $y_p = ax_p + b$ .

**Ex. 3.19.** coefficientAngulaire(d)

**Entrée :** Une droite d.

**Sortie :** Le coefficient angulaire de la droite d.

**Rappel :** a est le coefficient angulaire d'une droite  $y = ax + b$ .

**Ex. 3.20.** intersectionAbscisses(d)

**Entrée :** Une droite d.

**Sortie :** Le point intersection de d avec l'axe des abscisses.

**Rappel :**  $(-\frac{b}{a}, 0)$ .

**Ex. 3.21.** paralleles(d1, d2)

**Entrée :** Deux droites d1 et d2.

**Sortie :** True si d1 et d2 sont parallèles, False sinon.

**Rappel :** vérifier si les coefficients angulaires sont identiques.

**Ex. 3.22.** confondues(d1, d2)

**Entrée :** Deux droites d1 et d2.

**Sortie :** True si d1 et d2 sont confondues, False sinon.

**Rappel :** vérifier si les coefficients angulaires et les termes indépendants sont identiques.

**Ex. 3.23.** intersection(d1, d2)

**Entrée :** Deux droites d1 et d2.

**Sortie :** Un point d'intersection de d1 et d2 s'il existe, None sinon.

**Rappel :** Si les droites sont confondues, choisir un point quelconque d'une des droites. Si elles ne sont pas confondues mais parallèles, alors il n'y a pas d'intersection. Sinon, l'abscisse de l'intersection est donnée par  $x_{\cap} = \frac{b_2 - b_1}{a_1 - a_2}$  et l'ordonnée par  $y_{\cap} = a_1 x_{\cap} + b_1 = a_2 x_{\cap} + b_2$ .

**Ex. 3.24.** droiteNormale(d, p)

**Entrée :** Une droite d et un point p.

**Sortie :** La droite perpendiculaire à d passant par p.

**Rappel :**  $y = -\frac{x}{a} + (y_p + \frac{x_p}{a})$ .

**Ex. 3.25.** droiteParallele(d, p)

**Entrée :** Une droite d et un point p.

**Sortie :** La droite parallèle à d passant par p.

**Rappel :**  $y = ax + (y_p - ax_p)$ .

**Ex. 3.26.** distanceDroite(d, p)

**Entrée :** Une droite d et un point p.

**Sortie :** La distance entre d et p.

**Rappel :** distance euclidienne entre p et le point  $p'$  qui est l'intersection entre la droite d et la droite normale à d passant par p.

**Ex. 3.27.** symetrieOrthogonale(d, p)

**Entrée :** Une droite d et un point p.

**Sortie :** Le point qui est l'image de p par la symétrie orthogonale d'axe d.

**Rappel :** Soit  $p'$  l'intersection entre la droite d et la droite perpendiculaire à d passant par p. Alors la symétrie orthogonale de p par d est le point  $p'' = (x_p + 2(x_{p'} - x_p), y_p + 2(y_{p'} - y_p))$ .



**Ex. 3.28. \*\* (pour les férus de maths)** Vous pouvez généraliser ces fonctions à des droites quelconques dans le plan euclidien, mêmes verticales, en supposant plutôt que les droites sont représentées par un triplet  $(a, b, c)$  de réels qui correspondent à une équation  $ax + by + c = 0$ . Écrivez dès lors les fonctions précédentes en tenant compte de la possibilité d'avoir des droites verticales et horizontales.

# Chapitre 4

## Chaînes de caractères

### Exercices préparatoires

**Prép. 4.1.** Pourquoi les instructions suivantes ne donnent-elles pas le même résultat ?

```
"0199" < "187"
int("0199") < int("187")
```

**Prép. 4.2.** Que fait l'instruction suivante ?

```
"Tim" in "Castle Aaaaargh"
```

**Prép. 4.3.** Que fait le code suivant ?

```
txt = "Antioch"
for i in range(len(txt)):
 print(txt[i])
```

**Prép. 4.4.** Que fait le code suivant ?

```
for k in "Caerbannog":
 print(k, end=k)
```

**Prép. 4.5.** Expliquez pourquoi la seconde instruction du code suivant provoque une erreur.

```
msg = "Caerbannog"
msg[0] = "K"
```

**Prép. 4.6.** Quelle est la différence entre la chaîne vide "" et None ?

**Prép. 4.7.** En supposant que le fichier `test.txt` existe et est bien accessible, que fait le code suivant ? Expliquez pourquoi on fait appel à la méthode `strip()`.

```
fd = open("test.txt")
for i in fd:
 print(i.strip())
fd.close()
```

**Prép. 4.8.** En supposant que le fichier `test.txt` existe et est bien accessible, que fait le code suivant ?

```
fd = open("test.txt")
s = fd.readlines()
fd.close()
for i in s:
 print(i.strip())
```

**Prép. 4.9.** En supposant que le fichier `test.txt` existe et est bien accessible, que fait le code suivant ?

```
fd = open("test.txt")
s = fd.read()
while s != '':
 print(s.strip())
 s = fd.read()
fd.close()
```

**Prép. 4.10.** Expliquez, en termes de consommation de mémoire, quelles méthodes d'accès de fichier parmi celles montrées aux préparations 4.7 à 4.9 sont efficaces et inefficaces.

## Exercices en séance

**Ex. 4.1.** Écrivez les fonctions suivantes sous la forme d'un module appelé `stringmanip`. Dans tous les cas de figure, vos fonctions doivent admettre des indices positifs et négatifs.

1. Une fonction `caractere(s, i)` qui renvoie le  $i^{\text{e}}$  caractère de la chaîne de caractères `s`. Si ce caractère n'existe pas, la fonction renvoie la chaîne vide.
2. Une fonction `caracteres(s, i, j)` qui renvoie une chaîne de caractères contenant les caractères compris entre la position `i` et la position `j` (bornes incluses). Si ces bornes sont invalides, la fonction renvoie la chaîne vide.
3. Une fonction `change_caractere(s, i, a)` qui renvoie une chaîne de caractères identique à `s` dans laquelle le  $i^{\text{e}}$  caractère a été remplacé par le caractère stocké dans `a`. Si cette position est invalide ou que `a` contient plus d'un caractère, la fonction renvoie la chaîne vide.
4. Une fonction `change_caracteres(s, i, j, t)` qui renvoie une chaîne de caractères identique à `s` dans laquelle les caractères situés entre la position `i` et `j` (bornes incluses) ont été remplacés par la chaîne de caractères stockée dans `t`. Si les bornes `i` et `j` sont invalides ou que la chaîne `t` est trop courte ou trop longue, la fonction renvoie la chaîne vide.
5. Une fonction `trouve_caractere(s, a)` qui parcourra la chaîne `s` lettre par lettre et renverra la position (positive) de la première lettre identique à `a` rencontrée dans la chaîne. S'il n'y a aucune lettre `a` dans la chaîne de caractère, la fonction renvoie `-1`.

**Ex. 4.2.** Écrivez les fonctions suivantes (prenant toutes une chaîne de caractères en paramètre) sous la forme d'un module appelé `userinput` :

1. `convert_to_int` qui, si la chaîne de caractères représente un nombre entier, effectue la conversion et renvoie l'entier correspondant. Si cette chaîne représente tout autre chose, la fonction affichera un message d'erreur.

*Conseil :* utilisation de la méthode `isdigit()` appartenant au type chaîne de caractères.

2. `convert_to_float` qui, si la chaîne de caractères représente un nombre réel, renvoie le nombre réel. Si cette chaîne représente tout autre chose, la fonction affichera un message d'erreur. Vous ne pouvez pas utiliser la fonction `float()`.

*Conseil :* Pour rappel, un nombre entier est aussi un nombre réel dont la partie décimale est égale à 0. Les autres nombres réels (qui ne sont pas entiers) peuvent s'écrire sous la forme *partie\_entière.partie\_décimale* (pensez à la méthode `split()` sur les chaînes de caractères).

3. `is_one_word` est une fonction qui prend une chaîne de caractères et qui, si celle-ci contient un seul mot, renvoie `True`. Sinon, la fonction renvoie `False`.
4. `is_one_letter` est une fonction prenant une chaîne de caractères en paramètre et qui renvoie `True` si la chaîne de caractères représente une seule lettre. Sinon, la fonction renvoie `False`.
5. `belongs_to_dictionary` est une fonction prenant en paramètre une chaîne de caractères et vérifie si elle représente un mot qui figure dans la liste des mots contenus dans le fichier `words.txt`, auquel cas elle renverra `True`. Sinon, la fonction renvoie `False`. Attention : les mots du fichier `words.txt` seront stockés ligne par ligne dans le fichier.

**Ex. 4.3.** Écrivez une fonction `borders` qui prend une chaîne de caractères en paramètre et renvoie une liste contenant les bords de la chaîne de caractères. On dit qu'un mot `u` est un bord de `w` (avec  $u \neq w$ ) si `u` est à la fois un préfixe strict (c'est-à-dire non vide) de `w` et un suffixe strict (c'est-à-dire non vide) de `w`. Si `w` n'a pas de bord, la fonction renvoie la chaîne vide. Exemple : '`a`' et '`abda`' sont des bords de '`abdabda`'. Le plus grand bord est '`abda`'.

**Ex. 4.4.** Écrivez une fonction `plus_grand_bord(w)` qui, étant donné un mot `w`, renvoie le plus grand bord de ce mot.

**Ex. 4.5.** Écrivez une fonction `anagrammes(v, w)` qui renvoie `True` si et seulement si les mots `v` et `w` sont des anagrammes, c'est-à-dire des mots qui comprennent les mêmes lettres mais pas nécessairement dans le même ordre. Par exemple, '`marion`' et '`romina`' sont des anagrammes.

**Ex. 4.6.** Écrivez une fonction `intersection(v, w)` qui calcule l'intersection entre deux chaînes de caractères `v` et `w`. On définit l'intersection de deux mots comme étant la plus grande partie commune à ces deux mots. Par exemple, l'intersection de « *programme* » et « *grammaire* » est « *gramm* ».

**Ex. 4.7.** Écrire une fonction `trans(text, replaceA, replaceB)` (`replaceA` est un tuple (`oldA`, `newA`) et `replaceB` est un tuple (`oldB`, `newB`)), qui reçoit trois paramètres, et renvoie le résultat de la transformation suivante : chaque occurrence du symbole `oldA` dans la chaîne `text` est remplacée par la chaîne `newA`, et chaque occurrence du symbole `oldB` est remplacée par la chaîne `newB`.

**Exemple :**

```
>>> print(trans('ABBAB', ('A', 'AB'), ('B', 'BA')))
>>> 'ABBABAABBA'
```

**Ex. 4.8.** La méthode de chiffrement par décalage est une des premières techniques utilisées pour chiffrer un message. On appelle aussi cette technique le « chiffre césarien ». Dans cette méthode, on décale chaque lettre du message de  $k$  unités. Par exemple, si  $k = 3$ , la lettre « A » devient « D » par chiffrement. De même, si l'on considère le mot « BONJOUR », nous obtenons

avec un décalage de trois unités le mot « ERQMRXU ». Le décalage est circulaire : un décalage de trois unités sur la lettre « X » a par exemple pour résultat la lettre « A ». Nous considérons pour cet exercice que les messages ne sont composés que de caractères alphabétiques majuscules et non accentués.

Pour effectuer ce genre de décalages, nous pouvons utiliser la correspondance entre lettres et leur valeur numérique dans un encodage particulier, par exemple l'ASCII. La Table 4.1 reprend la correspondance entre les majuscules de « A » à « Z » et leur valeur numérique dans l'encodage ASCII. Il est important de remarquer que les valeurs se suivent en séquence ; en d'autres termes, pour obtenir la valeur numérique correspondant à la lettre « D », il suffit d'ajouter 3 à la valeur numérique de la lettre « A ».

| Lettre | Valeur ASCII |
|--------|--------------|
| A      | 65           |
| B      | 66           |
| ⋮      | ⋮            |
| Y      | 89           |
| Z      | 90           |

TABLE 4.1 – Correspondances entre majuscules et leur valeur dans l'encodage ASCII.

La fonction `ord()` en Python permet d'obtenir, pour une lettre donnée passée en paramètre, sa valeur numérique dans l'encodage ASCII. Par exemple, `ord('A')` renverra la valeur 65. La fonction `chr()` prend quant à elle une valeur numérique de l'encodage ASCII et renvoie la lettre correspondante : `chr(67)` renverra la lettre « C ».

À l'aide de ces deux fonctions, écrivez les fonctions suivantes :

- `caesar(mot, k)` qui renvoie le résultat du décalage de  $k$  unités du mot `mot`, reçu en paramètre (une chaîne de caractères).
- `canonique(mot)` qui renvoie une version canonique d'un mot. On définit la version canonique d'un mot comme étant l'unique décalage de ce mot donnant un mot débutant par la lettre « A ».
- Expliquez comment il serait possible de déchiffrer un message chiffré et quelle information devrait être transmise entre l'expéditeur et le destinataire pour que ce dernier comprenne bien le message.

**Ex. 4.9.\* (Examen de janvier 2012)** Le *chiffre de Vigenère* est une version améliorée du chiffre césarien vu à l'exercice 4.8. Dans cet algorithme, la clef n'est plus un simple décalage dans l'alphabet mais est un texte à part entière de longueur arbitraire.

|               |   |   |   |   |   |   |   |
|---------------|---|---|---|---|---|---|---|
| Texte clair   | Z | E | D |   | C | A | B |
| Clef          | A | B | B | A | A | B | B |
| Texte chiffré | Z | F | E |   | C | B | C |

FIGURE 4.1 – Chiffrement par l'algorithme de Vigenère.

Pour chiffrer un texte clair à la main, on utilise la *table de Vigenère* dont un extrait est donné dans la Table 4.2. Supposons que la clef choisie soit « ABBA » et que nous voulions chiffrer le texte clair « ZED CAB ». Dans un premier temps, nous étendons la clef en la concaténant à elle-même jusqu'à ce qu'elle ait la même longueur que le texte clair (en tronquant les éventuels caractères

|   | A | B | C | D | E | ... | Z |
|---|---|---|---|---|---|-----|---|
| A | A | B | C | D | E | ... | Z |
| B | B | C | D | E | F | ... | A |
| C | C | D | E | F | G | ... | B |
| D | D | E | F | G | H | ... | C |
| E | E | F | G | H | I | ... | D |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮   | ⋮ |
| Z | Z | A | B | C | D | ... | Y |

TABLE 4.2 – Extrait de la table de Vigenère. Les lignes correspondent à une lettre du texte clair, les colonnes à une lettre de la clef, et les cases du tableau aux lettres correspondantes du texte chiffré.

superflus) ; la clef « ABBA » devient donc « ABBAABB » car le texte clair est de longueur 7. Si la clef était plus longue que le texte clair de longueur  $n$ , on la tronque pour ne garder que les  $n$  premiers caractères. Ensuite, pour chaque caractère du texte clair, nous consultons la ligne correspondante dans la table, dans la colonne du caractère de la clef à la même position, et nous en tirons une lettre qui sera la lettre correspondante dans le chiffré. La première lettre du texte clair, « Z », restera donc « Z » dans le chiffré car chiffrer par la lettre « A » revient à ne rien changer si nous consultons le tableau. La seconde lettre du texte clair, « E », se fera chiffrer par la lettre « B » ; la table nous indique que le caractère dans le chiffré correspondant sera donc « F ». La Figure 4.1 vous donne le chiffrement complet de « ZED CAB » par la clef « ABBA ». Nous vous demandons d’écrire une fonction `vigenere()` prenant deux paramètres : une chaîne de caractères `plaintext` ainsi qu’une chaîne de caractères `key`. Cette fonction devra lire et chiffrer le texte clair donné par `plaintext` en utilisant la clef fournie, et renvoyer le texte chiffré. Vous pouvez supposer que le texte clair ne contient que des lettres majuscules ainsi que des symboles de ponctuation et des espaces. Notez que si on rencontre autre chose que des lettres, on laisse le caractère intact dans le chiffré (voir comportement vis-à-vis d’un caractère espace en Figure 4.1).

Vous pouvez utiliser les fonctions `ord()` et `chr()` vues à l’exercice 4.8. Nous vous encourageons à écrire des fonctions supplémentaires si cela peut vous aider ou clarifier votre code. Nous vous demandons d’utiliser parcimonieusement la mémoire ; évitez par exemple si possible de construire la table de Vigenère en mémoire.

# Chapitre 5

## Listes

### Exercices préparatoires

**Prép. 5.1.** Quelle est la différence entre un tuple et une liste ? Pour vous aidez, exécutez le code suivant :

```
a = (4, 2, 1)
a[1] = 9
print(a[1])
b = [4, 2, 1]
b[1] = 9
print(b[1])
```

**Prép. 5.2.** Que réalise chaque ligne du code suivant :

```
a = []
a.append(1)
print(a)
a.append(2)
print(a)
a.append([3])
print(a)
a += [4, 5, 6, 7]
print(a)
print(a[1:3])
print(a[3:])
print(a[:2])
```

**Prép. 5.3.** (Examen de janvier 2012) Que donne le code suivant ?

```
x = [1]
x[1] = 2
print(x)
```

**Prép. 5.4.** (Examen de janvier 2012) Que donne le code suivant ?

```
def foo(v):
 v = [3, 4]
x = [0, 1, 2]
foo(x)
print(x)
```

**Prép. 5.5. (Examen de janvier 2012)** Que donne le code suivant ?

```
x = [0,1,2]
y = x
y[:] = [3,4]
print(x,y)
```

**Prép. 5.6.** Construisez une liste  $L$  contenant un entier quelconque compris entre 1 et 10. Multipliez le par 2, ajoutez 8 et divisez le par 2. Soustrayez le nombre que vous avez choisi au départ à ce nouveau nombre. Vérifiez que le résultat vaut 4.

**Prép. 5.7.** Expliquez pourquoi le programme suivant affiche deux fois le même résultat.

```
L1 = []
L2 = L1
L1.extend([1,2])
L2.extend([3,4])
print(L1, L2, sep='\n')
```

**Prép. 5.8.** Quel résultat affiche ce code ? Expliquez pourquoi.

```
L1 = []
L2 = L1
print(L1 == L2)
print(L1 is L2)
```

**Prép. 5.9.** Quel résultat affiche ce code ? Expliquez pourquoi.

```
L1 = []
L2 = []
print(L1 == L2)
print(L1 is L2)
```

**Prép. 5.10.** En supposant que  $L1$  et  $L2$  soient des listes vides distinctes, quel est le résultat des opérations suivantes ? Expliquez la différence entre les deux.

```
L1.append('hello')
L2.extend('hello')
```

## Exercices en séance

**Note :** Dans les exercices qui suivent, nous vous demandons de ne pas utiliser les fonctions suivantes, prédéfinies par Python pour manipuler les listes : `insert`, `count`, `remove`, `map`, `filter`, `enumerate`.

**Ex. 5.1.** Écrivez une fonction `bornes(nombres)` qui reçoit en paramètre une liste de nombres et renvoie un tuple comprenant le maximum et le minimum des valeurs entières présentes dans cette liste.

**Ex. 5.2.** Écrivez une fonction `my_pow` qui prend comme paramètres un nombre entier `max` et un nombre flottant `b` et qui renverra une liste contenant les `max` premières puissances de `b`, c'est-à-dire une liste contenant les éléments allant de  $b^0$  à  $b^{\text{max}-1}$ .



**Ex. 5.3.** Écrivez une fonction `prime_numbers` qui prend comme paramètre un nombre entier `nb` et qui renverra une liste contenant les `nb` premiers nombres premiers. Nous vous demandons de penser aux différents cas pouvant intervenir dans l'exécution de la fonction. Pour rappel, un nombre premier  $b$  est un entier naturel qui admet que deux diviseurs distincts entiers et positifs :  $b$  et 1. En d'autres termes,  $b$  est premier si il n'existe pas d'entiers naturels  $c$  et  $d$  différents de  $b$  et de 1 tel que  $b = c \times d$ . Mathématiquement, on peut définir un nombre premier  $b$  comme étant un nombre tel que

$$(b > 2) \wedge \left( \nexists a \cdot (1 < a < b) \wedge (b \bmod a = 0) \right)$$

**Ex. 5.4.** Écrivez une fonction `my_insert` qui prend une liste triée `L` et un entier `n` en paramètres. La fonction devra insérer `n` dans la liste tout en la maintenant triée.

**Ex. 5.5.** Écrivez une fonction `my_count` qui prend une liste `L` et un élément `e` en paramètres. La fonction doit renvoyer le nombre de fois que l'élément `e` apparaît dans la liste. N'oubliez pas de tester tous les cas possibles.

**Ex. 5.6.** Écrivez une fonction `my_remove` qui prend une liste `L` et un élément `e` en paramètre et qui effacera la première apparition de l'élément `e` dans la liste. N'oubliez pas de tester tous les cas possibles.

**Ex. 5.7.** Écrivez une fonction `my_map` qui prend une liste `L` et une fonction `f` en paramètres et qui renverra une nouvelle liste où un élément à la  $i^{\text{e}}$  position contiendra la valeur de retour de la fonction `f` appliquée au  $i^{\text{e}}$  élément de la liste `L`. À nouveau, n'oubliez pas de tester tous les cas possibles.

**Ex. 5.8.** Écrivez une fonction `my_filter` qui prend une liste `L` et une fonction `f` en paramètres. Cette fonction renverra une nouvelle liste constituée des éléments de la liste `L` pour lesquels la fonction `f` renvoie `True`.

**Ex. 5.9.** Écrivez une fonction `my_enumerate` qui prend une liste `L` en paramètre. Cette fonction renverra une liste de tuples à deux composantes. Chaque tuple devra avoir en premier élément l'indice  $i$  et en deuxième élément le  $i^{\text{e}}$  élément de la liste `L`. Veuillez tester votre programme et repérer clairement ses limites d'utilisation.

**Ex. 5.10.** Écrivez une fonction `reduce` qui prend en paramètres une liste `L`, une fonction `f` (à deux paramètres) et un élément `e`. La fonction devra être initialement appliquée à l'élément `e` et au premier élément de la liste `L`. Ensuite, il faudra successivement appliquer la fonction `f` sur le résultat du précédent appel de fonction et l'élément suivant de la liste.

*Exemple :* `reduce(somme, [1, 2, 3, 4], 0)` renverra `((((0 + 1) + 2) + 3) + 4)`

**Ex. 5.11.** Écrivez une fonction `my_print` qui prend en paramètres une liste `L` et un tuple `separator` de taille 3. Les éléments de la liste devront être des nombres entiers. La fonction devra renvoyer une chaîne de caractères contenant dans l'ordre :

1. le premier élément de `separator`;
2. chaque élément de la liste `L` en insérant, entre chaque élément, le deuxième élément de `separator`;
3. le troisième élément de `separator`.

*Exemple :* `>>> my_print([1, 2, 3, 4], ('[', '->', ' '))` affichera `[1->2->3->4]`

**Ex. 5.12.** Écrivez une fonction `my_invert` qui inverse (en place) l'ordre des éléments dans une liste qui lui est donnée en paramètre sans utiliser une autre structure telle qu'une autre liste.

*Exemple :* `my_invert([1, 2, 3, 4])` renverra `[4, 3, 2, 1]`

**Ex. 5.13.** On se donne une liste qui encode une séquence  $t$ . Chaque élément de cette liste est un tuple de deux éléments : le nombre de répétitions successives de l'élément  $x$  dans  $t$ , et l'élément  $x$  lui-même. Les éléments successifs répétés forment la séquence  $t$ .

Écrivez une fonction `decompresse`, qui reçoit une telle liste en paramètre et renvoie la séquence  $t$  sous forme d'une nouvelle liste.

Exemple :

```
>>> L = [(4, 1), (2, 2), (2, 'test'), (3, 3), (1, 'bonjour')]
>>> decompresse (L)
[1, 1, 1, 1, 2, 2, 'test', 'test', 3, 3, 3, 'bonjour']
```

## Chapitre 6

# Matrices

### Exercices préparatoires

**Prép. 6.1.** Écrivez une fonction `init_mat(m, n)` et une fonction `print_mat(M)`. La première construit une matrice d'entiers initialisée à la matrice nulle et de dimension  $m \times n$ . La seconde prend une matrice en paramètre et affiche son contenu.

**Prép. 6.2.** Qu'affiche `print_mat(M)` lorsqu'on exécute au préalable le code suivant :

```
L = [0, 0]
M = [L, L]
L[0] = 2
```

**Prép. 6.3.** Le code précédent montre que la matrice `M` contient la liste `L`. Par conséquent, toute modification de `L` modifie l'affichage réalisé par `print_mat(M)`. Écrivez une fonction `my_deepcopy(L)` qui renvoie une nouvelle liste contenant les mêmes valeurs que `L`. Cette fonction vous permettra d'inclure les différentes valeurs de `L` (et non d'inclure `L`) à la matrice `M`.

### Exercices en séance

**Ex. 6.1.** Écrivez une fonction `trace(M)` qui prend en paramètre une matrice `M` de taille  $n \times n$  et qui calcule sa trace de la manière suivante :

$$\text{tr}(A) = \sum_{i=1}^n A_{ii}$$

**Ex. 6.2.** Une matrice  $M = \{m_{ij}\}$  de taille  $n \times n$  est dite antisymétrique lorsque pour toute paire d'indices  $i, j$ , on a  $m_{ij} = -m_{ji}$ . Écrire une fonction booléenne qui teste si une matrice est antisymétrique.

**Ex. 6.3.** Écrivez une fonction `produitMatriciel(A, B)` qui calcule le produit matriciel  $C$  (de taille  $m \times \ell$ ) entre les matrices  $A$  (de taille  $m \times n$ ) et  $B$  (de taille  $n \times \ell$ ). Le produit matriciel se calcule comme suit :

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}$$

**Ex. 6.4.** Écrivez une fonction `xorMatriciel(A, B)` qui prend en paramètres deux matrices  $A$  et  $B$  de même dimensions (de taille  $m \times n$ ), et qui renvoie une matrice résultant  $C$  contenant les éléments de  $A$  xorés aux éléments de  $B$ . Cette fonction opère comme suit :

$$C_{ij} = A_{ij} \oplus B_{ij}, \forall i, j : 0 \leq i < n \text{ et } 0 \leq j < m$$

**Ex. 6.5.** Écrivez une fonction `rotationGauche(A)` qui prend en paramètre une matrice  $A$  sur laquelle elle effectue des rotations gauche sur chaque ligne. Le nombre de déplacements effectués par la rotation est en fonction du nombre de la ligne en question ; Autrement dit, la ligne  $i$  subit une rotation de  $i$  positions.

Comment adapteriez-vous cette fonction pour effectuer une rotation droite ?

**Ex. 6.6.\* (Examen de juin 2012)** Soit une matrice  $A$  de taille  $m \times n \times q$  contenant des entiers quelconques et une matrice  $P$  de même taille contenant des entiers compris entre 0 et  $m \times n \times q - 1$ , tous distincts. On vous demande de réorganiser la matrice  $A$  de la manière suivante : l'élément à l'indice  $(a, b, c)$  de la matrice  $A$  d'origine devra, après réorganisation, se trouver à la position numéro  $P[a][b][c]$  de la matrice  $A$ . La matrice  $P$  est appelée *matrice de permutation* des éléments de la matrice  $A$ . Chaque élément de la matrice  $P$  à l'indice  $(a, b, c)$  contient la position où se trouvera l'élément  $(a, b, c)$  de  $A$  après l'ensemble des permutations. On définit la position d'un élément comme l'indice qu'aurait cet élément dans un vecteur qui serait constitué des lignes de  $A$  mises côte-à-côte. Plus précisément, la Figure 6.1 vous montre les positions des différents éléments se trouvant dans une matrice  $A$  de dimension  $3 \times 3 \times 2$ . On y voit par exemple que l'élément  $A[0][2][1]$  (c'est-à-dire première ligne, troisième colonne, deuxième niveau de profondeur) contient l'entier 11 et est donc en position 11. L'élément  $A[1][1][0]$  quant à lui est à la position 4.

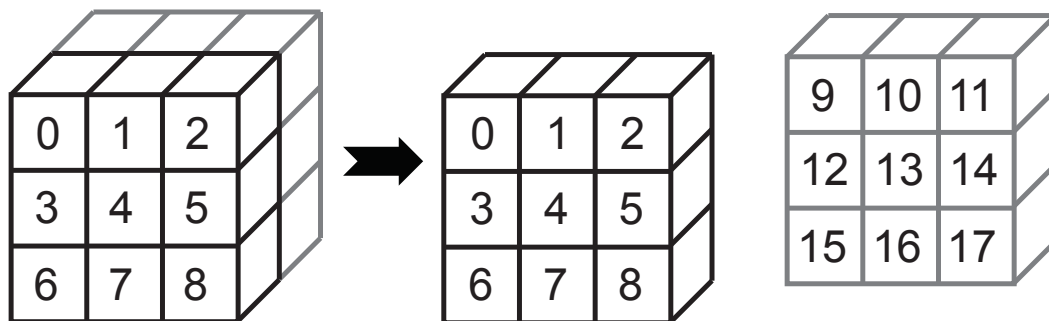


FIGURE 6.1 – Positions des différents éléments dans une matrice cubique.

On vous demande d'écrire un programme en Python qui permute les éléments d'une matrice  $A$  en fonction d'une matrice de permutation  $P$ . Le traitement doit être effectué sans utiliser de structure intermédiaire (c'est-à-dire pas de dictionnaire, pas de nouvelle liste ou de matrice, etc.). En d'autres mots, vous avez le droit de modifier la matrice  $A$ , la matrice  $P$  ainsi que d'utiliser des variables ne contenant que des entiers.

La Figure 6.2 vous montre un exemple de matrice  $A$ , de matrice  $P$  et enfin d'une matrice  $A'$  qui est la matrice  $A$  après permutation des éléments grâce à la matrice  $P$ . On y voit que l'entier 9 en position  $(0, 2, 1)$  dans la matrice  $A$  (i.e.  $A[0][2][1]$ ) doit se trouver en position 0 dans la matrice  $A$  après réorganisation (donc en  $A[0][0][0]$ ).

La Figure 6.3 vous affiche un autre exemple de matrice  $A$ , d'une matrice  $P$  et enfin d'une matrice  $A'$  qui est la matrice  $A$  après permutation des éléments grâce à la matrice  $P$ . Les flèches

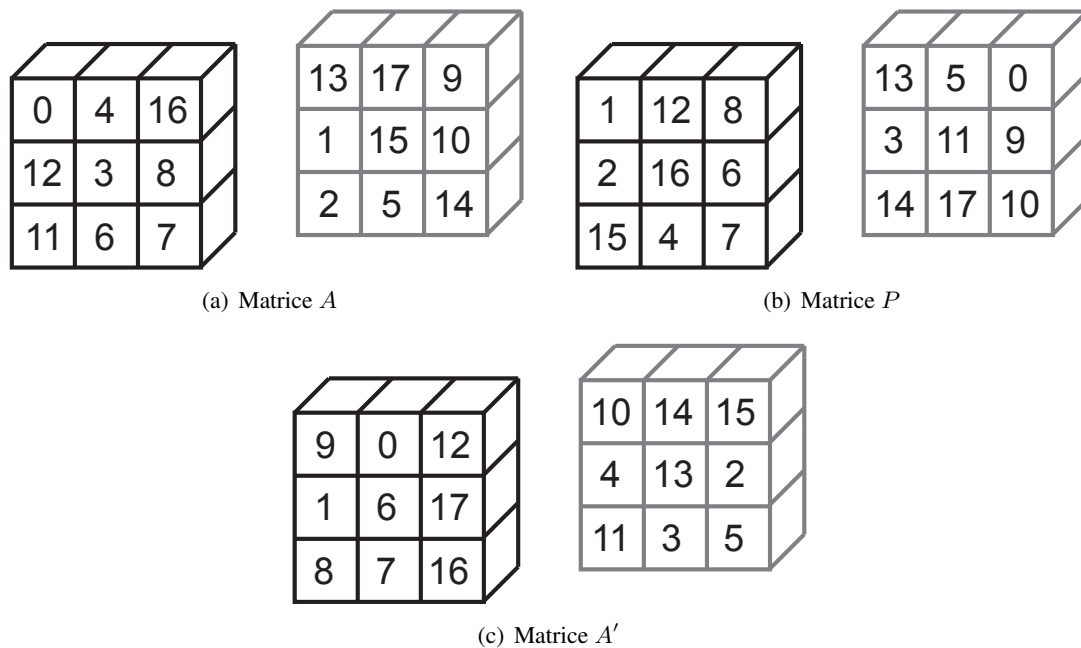


FIGURE 6.2 – Exemple de matrice  $A$  (figure a), de matrice  $P$  (figure b) et de matrice  $A'$  (figure c) représentant l'application de la permutation  $P$  sur  $A$ .

misés dans la Figure 6.3(a) vous permettent de voir comment sont permutées les valeurs dans la matrice  $A$ . On y voit que :

1. la valeur de  $A$  à l'indice 0 est placée dans la case de  $A'$  à l'indice 4,
2. puis la valeur de  $A$  à l'indice 4 est placée dans la case de  $A'$  à l'indice 2,
3. puis la valeur de  $A$  à l'indice 2 est placée dans la case de  $A'$  à l'indice 0,
4. puis la valeur de  $A$  à l'indice 1 est placée dans la case de  $A'$  à l'indice 3,
5. puis la valeur de  $A$  à l'indice 3 est placée dans la case de  $A'$  à l'indice 1,
6. et enfin la valeur de la dernière case n'est pas déplacée.

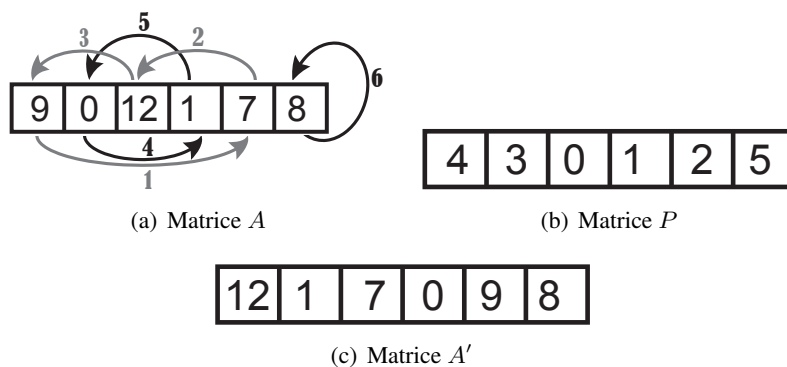


FIGURE 6.3 – Exemple de matrice  $A$  (figure a), de matrice  $P$  (figure b) et de matrice  $A'$  (figure c) représentant l'application de la permutation  $P$  sur  $A$ .

**Ex. 6.7.\*** Écrivez une fonction `rotation(M)` qui effectue une rotation, en place, de  $+90^\circ$  (dans le sens trigonométrique) d'une matrice carrée  $M$  de taille  $n \times n$ .

**Ex. 6.8. (mini-projet)** Le jeu du Morpion est un jeu dans lequel s'opposent deux joueurs. Le but dans ce jeu est d'aligner 5 fois un même jeton dans la grille de taille  $5 \times 5$  de départ. Il vous est demandé d'implémenter une version du Morpion en suivant les consignes suivantes :

- Créez une fonction `create_morpion(n)` qui crée et renvoie une grille de taille  $n \times n$  sur laquelle les joueurs seront amenés à jouer.
- Créez une fonction `next_turn(morpion, player, i, j)` qui simule l'ajout d'un jeton en position  $i, j$  par le joueur `player` (1 ou 2). Cette fonction renvoie une nouvelle grille correspondant à l'ajout du jeton sur la grille morpion passée en paramètre. Si le coup joué n'est pas valide (mauvaises positions, position déjà complète, etc.), cette fonction renvoie explicitement `None`.
- Créez une fonction `finished(morpion, i, j)` qui renvoie  $-1$  si la partie n'est pas terminée,  $0$  si la grille correspond à une situation de match nul,  $1$  si le joueur ayant placé le dernier pion a gagné la partie. Un joueur emporte la partie s'il y a une ligne, une colonne ou une (des deux) diagonale qui ne contient que des jetons de ce joueur.
- Créez une fonction `play_morpion()` qui demande à l'utilisateur la taille de la grille, et propose à deux joueurs d'effectuer une partie du Morpion.

# Chapitre 7

## Dictionnaires

### Exercices préparatoires

Matière à réviser :

- syntaxe du dictionnaire (`d[clef] = valeur`);
- fonctions `len()`, `values()`, `keys()`, `items()`, `clear()`, `copy()`, `get()` ;
- opérateur `in`;
- l'exercice 4.8 (p. 19).

**Prép. 7.1.** Parmi les instructions suivantes, quelles sont celles qui engendrent une erreur ? Sans ces instructions erronées, que contient le dictionnaire `d` à la fin ?

```
d = {}
d[1] = 22
d[3.14] = 24
b = 4.5
d[b] = 90
d['b'] = 25
d[(1,2)] = 'X'
d[[3,4]] = 32
d[2] = (3,4)
d[5,4] = [5,4]
```

**Prép. 7.2.** Devinez le résultat de l'opération suivante :

```
sum(set([1,1,1,2,1,1,2,3,5,8]))
```

Expliquez.

### Exercices en séance

**Ex. 7.1.** Dans un texte, il nous arrive souvent de remplacer des mots par des abréviations (exemple : « bonjour » par « bjr »). Nous allons utiliser un dictionnaire pour contenir les abréviations et leur signification.

Nous vous demandons d'écrire une fonction `remplacer(message, abbreviation)` qui va créer une copie de la chaîne de caractères `message` dans laquelle les mots qui figurent dans le dictionnaire `abbreviation` (comme clé) sont remplacés par leur signification (valeur).

Exemple d'utilisation :

```

abbrev = {'C.' : 'Chuck',
 'N.' : 'Norris',
 'cpt' : 'counted',
 '2' : '2 times',
 'inf' : 'infinity'}
print(remplacer('C. N. cpt 2 to inf', abbrev))

```

**Ex. 7.2.** Écrivez trois fonctions :

- addition(a,b) : qui renvoie la somme de a et b
- soustraction(a,b) : qui renvoie la différence entre a et b
- multiplication(a,b) : qui renvoie le produit de a et b

Nous vous demandons aussi de rédiger un script permettant à l'utilisateur d'agir avec ces fonctions selon son désir :

```

Rentrer deux valeurs :
a : 5
b : 6
Choisissez :
- (A)dditionner a et b
- (M)ultiplier a par b
- (S)oustraire a à b
- (Q)uitter

```

Suivant la lettre entrée au clavier par l'utilisateur pour son choix (clé), on effectuera alors l'appel de la fonction correspondante en la sélectionnant dans un dictionnaire de fonctions (valeurs).

**Ex. 7.3.** Reprenez l'exercice 4.8 (p. 19) et écrivez une fonction `equiv_canonique()` prenant un nombre arbitraire de mots en paramètres. Votre fonction devra renvoyer un dictionnaire où chaque clef est une forme canonique et où la valeur correspondante est la sous-liste des mots passés en paramètres respectant cette forme. Par exemple,

```

equivalence_canonique('BBB', 'CCC', 'BABA', 'ABBA')

```

devra renvoyer un dictionnaire de la forme :

```

{'AAA': ['BBB', 'CCC'], 'AAZ': ['BABA'], 'ABBA': ['ABBA']}.

```

**Ex. 7.4.** Écrivez une fonction `values(dico)` qui doit fournir, à partir du dictionnaire donné en paramètre, une liste des valeurs du dictionnaire telles qu'elles sont triées sur base de la clef du dictionnaire. Vous pouvez supposer que les clefs du dictionnaire sont des strings et vous pouvez utiliser la méthode `sort()` sur les listes pour vous aider.

**Ex. 7.5.** Écrivez une fonction `primesOddsNumbers(numbers)` qui reçoit une liste de nombres et qui affiche les nombres impaires de cette liste ainsi que les nombre premiers. Votre fonction doit faire appel à deux autres fonctions (que vous ne devrez pas écrire) qui renvoient, respectivement, un ensemble de nombres premiers et un ensemble de nombre paires.

**Ex. 7.6.** Écrivez une fonction `store_email(liste_mails)` qui reçoit en paramètre une liste d'adresses e-mail et qui renvoie un dictionnaire avec comme clefs le domaine des adresses e-mail et comme valeurs les listes d'utilisateurs correspondantes. Par exemple, la liste d'adresses suivante :

```

["jvervier@lit.ulb", "andre.colon@stud.ulb", "thierry@profs.ulb",
 "info@lit.ulb", "eric.ramzi@stud.ucl", "bernard@profs.ulb",
 "jean@profs.ulb"]

```

donnerait le dictionnaire suivant :



```
{ "lit.ulb" : ["jvervier", "info"]
 "stud.ulb" : ["andre.colon"]
 "profs.ulb" : ["thierry", "bernard", "jean"]
 "stud.ucl" : ["eric.ramzi"] }
```

**Ex. 7.7. (mini-projet)** Un dictionnaire peut nous permettre de stocker un tableau partiel, c'est-à-dire uniquement les cases remplies du tableau avec comme valeur le contenu de la case en question. Voici un exemple :

|  |   |   |   |  |   |
|--|---|---|---|--|---|
|  |   |   |   |  |   |
|  | X |   | O |  |   |
|  |   |   | O |  |   |
|  |   | O |   |  |   |
|  |   |   |   |  | O |
|  |   |   |   |  |   |

X : trésor  
O : piège

FIGURE 7.1 – Exemple de tableau

La tableau ci-dessus peut être implémenté de la manière suivante :

```
MY_PREVIOUS = 1
TRAP = -1
map = {}
map[(1,1)] = MY_PREVIOUS
map[(2,3)] = TRAP
map[(4,5)] = TRAP
map[(1,3)] = TRAP
map[(3,2)] = TRAP
```

On vous demande d'implémenter les fonctions suivantes :

- `createMap(size, trapsNbr)` qui reçoit en paramètres deux nombres naturels `size` et `trapsNbr` supérieurs ou égaux à 2 et qui renvoie un dictionnaire représentant une carte de taille `size × size` dans laquelle on place `trapsNbr` pièges et un trésor de manière aléatoire (utilisez le module `random`).
- `playGame(mapSize, trapsNbr)` qui crée une carte de taille `mapSize × mapSize` dans laquelle `trapsNbr` pièges et un trésor ont été placés de manière aléatoire et qui demande à l'utilisateur d'entrer une coordonnée. La fonction doit demander à répétition des coordonnées jusqu'à ce que l'utilisateur trouve le trésor (gagné) ou tombe sur un piège (perdu).

**Ex. 7.8.\* (Examen de juin 2011)** Dans les sciences expérimentales ainsi que dans bien d'autres domaines, on ne peut pas simplement se contenter d'effectuer des mesures mais il est également important de pouvoir caractériser celles-ci, comme par exemple via le calcul de moyenne, médiane, variance, etc. Un outil particulièrement utile dans ce contexte est l'*histogramme* dont le rôle est de pouvoir étudier la distribution de valeurs récoltées. Sa représentation la plus typique est sous forme d'un graphique en bâtons (voir Figure 7.2).

Étant donnée une liste  $L$  de valeurs (supposées réelles), nous pouvons représenter un histogramme par un dictionnaire qui associe, pour des gammes de valeurs données, le nombre de valeurs présentes dans  $L$  et ayant une valeur comprise dans cette gamme. Une application particulière d'un histogramme serait par exemple d'étudier les points d'étudiants à un examen donné ; l'histogramme pourrait alors décrire combien d'étudiants ont eu une note entre 0 et 4,

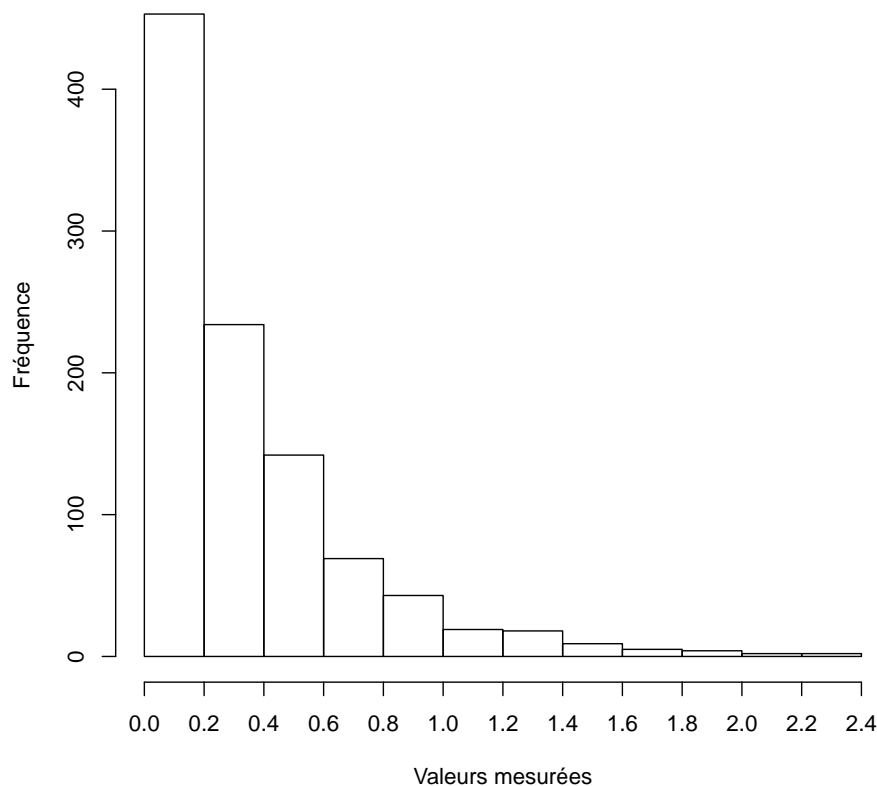


FIGURE 7.2 – Exemple d’histogramme illustré sous forme graphique sur base d’une liste de 1000 valeurs. Il peut par exemple être vu que plus de 400 valeurs mesurées se trouvaient dans l’intervalle  $[0.0, 0.2)$ .

entre 5 et 9, entre 10 et 14 et entre 15 et 20 par exemple. Notez que le nombre de catégories (c’est-à-dire de gammes de valeurs) est arbitraire.

Nous vous demandons d’écrire une fonction `histogramme()` qui prendra deux paramètres : une liste `L` de valeurs réelles (éventuellement négatives) ainsi qu’une liste `I` d’au moins deux valeurs réelles distinctes et triées dans l’ordre croissant. La liste `I` donne les bornes d’intervalles successifs sur la droite réelle. Par exemple, si cette liste est `[-5, 2, 9.5]`, elle symbolise deux intervalles : les valeurs entre  $-5$  (inclus) et  $2$  (non inclus) ainsi qu’entre  $2$  (inclus) et  $9.5$  (non inclus).

Votre fonction `histogramme()` doit renvoyer un dictionnaire où chaque clef est un string de la forme `"[a, b)"` si l’on suppose que  $a$  est la borne inférieure (incluse) de l’intervalle et que  $b$  est la borne supérieure (non incluse) ; la valeur associée à chaque clef sera un entier qui dira combien de valeurs de la liste `L` sont incluses dans l’intervalle considéré.

Dans un second temps, nous vous demandons d’écrire une fonction `intervalles()` qui prendra trois paramètres : deux réels `inf` et `sup` (il est supposé que  $\text{sup} > \text{inf}$ ) ainsi qu’un entier strictement positif `n`. La fonction devra renvoyer une liste triée tel qu’utilisée par la fonction `histogramme()` et qui correspond à `n` intervalles de même taille telles que la borne

inférieure du premier intervalle est `inf` et que la borne supérieure du dernier intervalle est `sup`.  
Un exemple possible d'exécution (en mode interactif) serait le suivant :

```
>>> L = [-1, 0, 3, -10, 0.5, 9, 8, 12]
>>> I = intervalles(-7, 9.5, 4)
>>> histogramme(L, I)
{ '[-7,-2.875)': 0, '[-2.875,1.25)': 3,
 '[1.25,5.375)': 1, '[5.375,9.5)': 2 }
```

Vous ne pouvez pas utiliser la fonction `sort()` pour vous aider.

# Chapitre 8

## Tris

### Choses à revoir

Revoir les tris suivants :

1. tri par énumération
2. tri bulle

### Exercices préparatoires

**Prép. 8.1.** Triez à la main la liste suivante avec le tri par énumération, et puis avec le tri bulle :

$$V = \begin{array}{|c|c|c|c|c|c|c|} \hline 4 & 8 & 4 & 2 & 8 & 1 & \\ \hline \end{array}$$

### Exercices en séance

**Ex. 8.1.** Écrire une fonction triant un tableau par énumération (les valeurs possibles pour les composantes :  $\{0, \dots, 10\}$ ).

Pour rappel, le tri par énumération fonctionne comme suit, en supposant qu'on désire trier le tableau  $V$  :

1. On commence par *compter* le nombre d'occurrences de chaque valeur du tableau  $V$  et stocker ces valeurs dans un tableau `count`. À la fin de cette étape, `count[i]` contiendra le nombre d'occurrences de la valeur  $i$  dans  $V$ .
2. On somme ensuite les valeurs contenues dans `count` de manière à ce que `count[i]` contienne le nombre de valeurs  $\leq i$  présentes dans  $V$ .
3. `count` permet alors de recopier les valeurs de  $V$  dans un nouveau tableau  $W$ , et ce, de manière triée. Pour ce faire, on parcourt  $V$  élément par élément. Chaque valeur  $v$  présente dans  $V$ , doit être placée dans la  $(\text{count}[v] - 1)^{\text{e}}$  case de  $W$ , et `count[v]` doit être décrémentée pour éviter les collisions.

**Ex. 8.2.** Implémentez le tri bulle. Cet algorithme fait avancer les plus grands éléments vers la fin du tableau au fur et à mesure de l'exécution de l'algorithme, à l'image de bulles dans un verre de Perrier remontant à la surface (la surface correspondant au côté droit du tableau).

À chaque itération, l'algorithme procède à des échanges successifs entre valeurs adjacentes : si une valeur en position  $i$  est plus grande que celle en  $i + 1$ , elles sont échangées. À la fin de la première itération de l'algorithme, nous avons dès lors le maximum du tableau dans la dernière

case. À chaque itération suivante, l'algorithme placera le maximum du reste du tableau dans la case juste avant le maximum précédent. Au bout de  $n$  itérations, le tableau sera dès lors bien trié.

**Ex. 8.3.** Implémentez le tri « *cocktail shaker* », variante du tri bulle. À chaque itération, l'algorithme fait d'abord en sorte qu'un maximum se retrouve le plus à droite possible du tableau. Deplus, il fait aussi en sorte qu'un minimum non encore traité se retrouve le plus à gauche possible. Une itération de l'algorithme place donc deux éléments à leur place : un maximum et un minimum.

**Ex. 8.4.** Implémentez le « *shotgun sort* ». Cet algorithme de tri se contente de mélanger aléatoirement tout un vecteur et de vérifier ensuite si le résultat est bien trié. Si ce n'est pas le cas, on recommence ces deux opérations. Pensez à utiliser la fonction `shuffle()` du module `random`. Quelles sont les complexités au meilleur et au pire cas de cet algorithme en sachant qu'un mélange se fait en  $\mathcal{O}(n)$  ?

**Ex. 8.5.** Implémentez le tri « *gnome* » qui fonctionne comme suit en partant de l'élément en position 0 jusque  $n - 2$  compris :

1. si  $V[i] \leq V[i + 1]$ , incrémenter  $i$  ;
2. sinon, échanger  $V[i]$  et  $V[i + 1]$  et décrémenter  $i$  si  $i > 0$ .

**Ex. 8.6.** Implémentez une étape de « *merge sort* ». Cet algorithme reçoit en paramètre deux listes triées :  $L_1$  et  $L_2$ . Le résultat est une liste triée *result* qui contient tous les éléments de  $L_1$  et de  $L_2$ . Note : il ne suffit pas de concaténer les deux listes.

**Ex. 8.7.\*\*** Le tri « *quicksort* » consiste à placer un élément de la liste  $L$  (appelé pivot, qu'on prend par défaut comme le dernier élément de la liste) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite. Cette opération s'appelle le partitionnement. Pour chacune des sous-listes, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

Implémentez le tri « *quicksort* » décrit dans l'algorithme suivant. La fonction a la signature suivante : `quicksort(L, debut, fin)` où  $L$  est une liste d'éléments comparables, `debut` et `fin` sont des entiers correspondant à des indices dans la liste.

---

```

if debut < fin then
 pivot = partitionner(L,debut,fin)
 quicksort(L,debut,pivot-1)
 quicksort(L,pivot+1,fin)
end if

```

---

# Chapitre 9

## Complexité

### Exercices préparatoires

Réviser le chapitre complexité du cours.

**Prép. 9.1.** Trouvez la complexité au pire cas du code suivant :

```
i = 0
while i < n :
 t[i] = 0
 i += 3
```

**Prép. 9.2.** Trouvez la complexité au pire cas de la fonction suivante (a, b et c sont des vecteurs de taille  $n*n$ ) :

```
def somme(c,a,b,n):
 for i in range(n):
 for j in range(n):
 c[i][j] = a[i][j] + b[j][i]
```

**Prép. 9.3.** Trouvez la complexité au pire cas du code suivant :

```
for j in range(4):
 for i in range(n):
 t[i] *= t[i]
```

**Prép. 9.4. (Examen de janvier 2010)** Donnez la complexité des opérations suivantes sur une liste Python en justifiant vos réponses.

- accéder à un élément d'un indice donné ( $x = t[i]$ )
- ajouter un élément en fin de liste ( $t.append(3)$ )
- insérer un élément ( $t.insert(0, 8)$ )
- supprimer un élément ( $t.remove(4)$ )
- copier une liste
- concaténer 2 listes
- rechercher un élément dans une séquence ( $x$  **in**  $s$ )

## Exercices en séance

**Ex. 9.1.** Trouvez la complexité au pire cas de la fonction suivante ( $a$ ,  $b$  et  $c$  sont des vecteurs de taille  $n*n$ ):

```
def produit(c,a,b,n):
 for i in range(n):
 for j in range(n):
 c[i][j] = 0
 for k in range(n):
 c[i][j] += a[i][k] * b[k][j]
```

**Ex. 9.2.** Trouvez la complexité au pire cas du code suivant :

```
i = 0
j = 0

while j < n:
 t[j*n+i] = i
 if i == n-1:
 j += 1
 i = 0
 else:
 i += 1
```

**Ex. 9.3.** En faisant l'hypothèse que  $n$  est positif, donnez la complexité au pire cas du code suivant :

```
c = 0

i = n * n
while i > 0:
 j = n
 while j > 0:
 c += 1
 j = j//4
 i = i//2
print c
```

**Ex. 9.4.** Trouvez la complexité au pire cas de la fonction suivante ( $a$  est un vecteur de taille  $n*n$ ):

```
def symetrie(a,n):
 sym = True
 i = 0
 while i < n and sym:
 j = 0
 while j < i and sym:
 if a[i][j] != a[j][i]:
 sym = False
 j += 1
 i += 1
 return sym
```

**Ex. 9.5.\*** Trouvez les complexités au pire cas et minimale du code suivant, en supposant que la fonction `boolrand` renvoie `True` ou `False` avec la même probabilité et qu'elle est de complexité constante.

```

for i in range(n):
 v[i] = True
for i in range(n-1):
 w[i] = boolrand()
w[n-1] = False
flag = True
j = 1
while flag:
 if v[j] == w[j]:
 for i in range(j, n-1):
 w[i] = boolrand()
 else:
 flag = False
 j += 1

```

**Ex. 9.6.\* (Examen d'août 2010)** On se donne deux fonctions, `expobeta` et `expo`, qui calculent toutes les deux la valeur du paramètre  $x$  élevée à la puissance  $n$ . On suppose  $x$  et  $n$  strictement positifs, et on ignore les problèmes de débordement.

1. Donnez les complexités au pire et meilleur cas de ces deux fonctions sous la forme d'un grand  $\mathcal{O}$  en fonction du paramètre  $n$ .
2. La fonction `expo` est-elle plus efficace ? Expliquez.

```

def expobeta(x,n):
 x1 = x
 for i in range(1,n):
 x *= x1
 return x

```

```

def expo(x,n):
 x1 = x
 i = 1
 while 2*i <= n:
 x = x*x
 i = 2*i
 while i < n:
 x *= x1
 i += 1
 return x

```

**Ex. 9.7.\* (Examen d'août 2009)** Donnez la complexité au pire cas, en fonction de  $n$  et en utilisant la notation  $\mathcal{O}$ , des fonctions suivantes.

```

def f(V):
 n = len(V)
 i = 0
 j = 0
 while i < n:
 while j < i:
 V[j] *= i - j
 j += 1
 i += 3

```



```

def g(V):
 n = len(V)
 i = n - 1
 while i >= 0:
 V[i] = 0
 for j in range(i+1):
 V[i] += j
 i -= 1

def h(V):
 n = len(V)
 for i in range(n):
 V[i] = 0
 j = n
 while j > 1:
 V[i] += i*j
 j //= 2

```

**Ex. 9.8.\* (Examen de juin 2010)**

Donnez la complexité de la fonction `fuse`, en tenant compte des différentes opérations et en justifiant vos réponses.

```

def fuse(t1, t2):
 if len(t1) == 0:
 return t2
 elif len(t2) == 0:
 return t1
 elif t1[0] < t2[0]:
 return [t1[0]] + fuse(t1[1:], t2)
 else:
 return [t2[0]] + fuse(t1, t2[1:])

```

**Ex. 9.9.\* (Examen de janvier 2012)**

- Donnez la complexité du code suivant où  $n$  est un paramètre et  $mat$  une matrice de dimension  $n \times n$  dont chaque élément contient un string de taille  $n$  :

```

sum = 0
for i in range(n):
 for j in range(n):
 if "a" in mat[i][j]:
 sum = sum + 1

```

- Donnez la complexité du code suivant où  $n$  est un paramètre :

```

i = 2
while i < n ** n:
 i = i * i

```

**Ex. 9.10.\* (Examen de juin 2012)** Donnez la complexité au pire cas de l'algorithme suivant en fonction de N.

```
def incseq(V):
 N = len(V)
 i = 0
 k = 0
 maxseq = 0

 while i < N-1:
 seq = 1
 j = i
 done = False
 while j < N-1 and not done:
 if V[j] > V[j+1]:
 done = True
 else:
 seq += 1
 j += 1
 if seq > maxseq:
 maxseq = seq
 k = i
 i += seq

 return k
```

# Chapitre 10

## Logique et invariants

### Exercices préparatoires

Réviser la matière concernée vue au cours ainsi que la logique de base telle que vue au cours de mathématiques. Nous vous invitons également à déjà essayer de réaliser l'exercice 10.1 du mieux que vous pouvez.

### Rappels théoriques

Rappels de logique :

1.  $\nexists i \cdot \phi(i) \iff \forall i \cdot \neg \phi(i)$  où  $\phi$  est une formule logique fonction de  $i$  ;
2.  $a \rightarrow b \rightarrow c \iff (a \wedge b) \rightarrow c$  ;
3. Priorité des opérateurs logiques :  $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$  ;
4.  $\rightarrow$  est associatif à droite :  $a \rightarrow b \rightarrow c$  est équivalent à  $a \rightarrow (b \rightarrow c)$ .

Formes à retenir :

1.  $\forall i \cdot (0 \leq i < n) \rightarrow (\dots)$
2.  $\exists i \cdot (0 \leq i < n) \wedge (\dots)$

Propriétés à respecter pour qu'un invariant  $Inv$  d'une boucle de condition  $C$  soit correct :

1.  $Pre \{init\} \implies Inv$
2.  $Inv \wedge C \{boucle\} \implies Inv$
3.  $Inv \wedge \neg C \{fin\} \implies Post$

**Preuve partielle** : prouver que l'invariant d'une boucle est correct.

**Preuve de terminaison** : prouver qu'avec les préconditions et initialisations, la boucle se termine toujours.

**Preuve totale** : preuve partielle + preuve de terminaison.

### Exercices en séance

#### Logique du premier ordre

**Ex. 10.1.** Exprimez en logique du premier ordre les affirmations suivantes (vous supposerez que tous les vecteurs sont de taille  $n$ ) :

1. La variable  $v$  est positive ou nulle.
2. La variable  $v$  contient une valeur comprise entre 5 et 8 (bornes incluses).
3. Si la variable  $v$  est positive, alors la variable  $w$  est négative ou nulle.
4. Le vecteur  $V$  ne contient que des valeurs positives ou nulles.
5. Au moins une case du vecteur  $V$  contient une valeur supérieure ou égale à 5.
6. La variable  $v$  contient la somme des éléments du vecteur  $W$ .
7. La variable  $v$  contient la somme des  $i$  premiers éléments du vecteur  $W$ .
8. Les  $i$  premiers éléments du vecteur  $V$  sont inférieurs à 6.
9. Tous les éléments du vecteur  $V$  sont inférieurs ou égaux à  $\max$ .
10.  $\max$  est la valeur maximale contenue dans le vecteur.
11. Le vecteur  $V$  contient toutes des valeurs différentes.
12. Le vecteur  $V$  est trié (de façon croissante).
13. Les  $i$  premières cases du vecteur  $V$  sont triées (de façon décroissante).
14. Les cases comprises entre les indices  $k_1$  et  $k_2$  du vecteur  $V$  sont triées (de façon non-décroissante).
15. Si la variable  $i$  est positive, les  $i$  premières cases du vecteur  $V$  sont triées de façon non-décroissante ; sinon, le vecteur  $V$  est initialisé à 0.
16. La partie du vecteur  $V$  comprise entre les indices  $k_1$  et  $k_2$  est triée, et il n'existe pas de plus grande partie (en terme de nombre de cases) du vecteur qui soit triée.

**Ex. 10.2. \*** (Examen d'août 2010) Exprimez en logique du premier ordre que le vecteur  $V$  à  $n$  composantes entières d'indices  $\{0, 1, \dots, n-1\}$  :

1. est tel que toutes ses composantes (exceptée celle d'indice 0) ont une valeur qui est un multiple de la composante précédente.
2. est tel que chacune de ses composantes a une valeur qui est un multiple de son indice correspondant.

**Ex. 10.3. \*** (Examen de janvier 2010) Pour le code suivant, sachant que les valeurs lues dans la matrice  $M$  de dimensions  $\text{MAX} \times \text{MAX}$  sont toutes dans l'intervalle  $[0, \text{MAX} \times \text{MAX} - 1]$ , exprimez en logique du premier ordre la condition sur le contenu de  $M$  pour que le code renvoie un résultat vrai. Donnez aussi la complexité au pire cas de l'algorithme. Justifiez vos réponses.

```
def condition(M):
 MAX = len(M)
 V = [False] * (MAX*MAX)
 for i in range(MAX):
 for j in range(MAX):
 V[M[i][j]] = True
 n = 0
 while n < MAX*MAX and V[n]:
 n += 1
 return n == MAX*MAX
```

**Ex. 10.4. \*** (Examen de janvier 2010) Pour le code suivant, sachant que les valeurs lues dans la matrice  $M$  de dimensions  $\text{MAX} \times \text{MAX}$  sont toutes dans l'intervalle  $[0, \text{MAX} \times \text{MAX} - 1]$ , exprimez en logique du premier ordre la condition sur le contenu de  $M$  pour que le code renvoie un résultat vrai. Donnez aussi la complexité au pire cas de l'algorithme. Justifiez vos réponses.

```

def condition(M) :
 MAX = len(M)
 found = True
 k = 0
 while k < MAX*MAX and found:
 found = False
 i = 0
 while i < MAX and not found:
 j = 0
 while j < MAX and not found:
 found = (M[i][j] == k)
 j += 1
 i += 1
 k += 1
 return found

```

**Ex. 10.5.\* (Examen de juin 2010)** Soit une liste d'entiers  $V$  à  $n$  éléments ( $n$  strictement positif). Exprimez en logique du premier ordre que

1.  $V$  ne contient pas deux éléments de valeur égale ;
2.  $V$  est tel que ses éléments de **valeur** paire sont triés par ordre croissant et ses éléments d'**indice** impair sont triés par ordre décroissant.

**Ex. 10.6.\* (Examen de juin 2010)** Pour le code suivant, où  $n$  est une constante entière strictement positive et en ayant des valeurs entières quelconques dans la matrice  $M$  (carrée de côté  $n$ ), exprimez en logique du premier ordre la condition sur le contenu de  $M$  pour que le code renvoie un résultat vrai. Donnez également la complexité au pire cas de l'algorithme.

```

def condition(M) :
 n = len(M)
 found = True
 i = 0
 while found and i < n:
 found = False
 j = 0
 while not found and j < n-1:
 k = j+1
 while not found and k < n:
 found = (M[i][j] == M[i][k])
 k += 1
 j += 1
 i += 1
 return found

```

**Ex. 10.7.\* (Examen d'août 2009)** La fonction `test` ci-dessous reçoit un vecteur en paramètre et vérifie si celui-ci satisfait une certaine propriété. Donnez, d'abord en français puis sous forme d'une formule logique, la propriété testée. En d'autres termes, exprimez une condition nécessaire et suffisante sur le vecteur pour que la fonction `test` renvoie la valeur `True`.

```

def test(V) :
 n = len(V)
 stop = False
 if n < 3:
 return True
 diff = V[1] - V[0]

```

```

i = 1

while not stop and i < n-1:
 nvdiff = V[i+1] - V[i]
 stop = (diff > nvdiff)
 diff = nvdiff
 i += 1

return not stop

```

**Ex. 10.8.\* (Examen de juin 2011)** L'algorithme suivant fusionne 2 listes triées.

1. Exprimez en logique des prédicats, les préconditions (hypothèses sur les paramètres) et la postcondition (résultats) de la fonction `fuse`.
2. Donnez la complexité (au pire cas) de la fonction `fuse`, en tenant compte des différentes opérations et en justifiant vos réponses (sans justification, la réponse est nulle).

```

def fuse(t1, t2):
 if len(t1) == 0:
 return t2
 elif len(t2) == 0:
 return t1
 elif t1[0] < t2[0]:
 return [t1[0]] + fuse(t1[1:], t2)
 else:
 return [t2[0]] + fuse(t1, t2[1:])

```

### Invariants

**Ex. 10.9.** Vérifiez que  $(0 \leq i \leq n) \wedge (j = i \times m)$  est un invariant correct pour la boucle suivante (on suppose que  $m$  et  $n$  sont positifs) :

```

i = 0
j = 0
while i != n:
 j += m
 i += 1

```

**Ex. 10.10.** Soit le morceau de code suivant (on suppose que  $n$  est strictement positif et que  $V$  et  $W$  sont des vecteurs d'entiers de dimension  $n$ ) :

```

i = 0
s = 0
while i < n:
 s += V[i]*W[i]
 i += 1

```

1. Expliquez en français ce que fait cette boucle.
2. Exprimez à l'aide d'une formule logique les précondition et postcondition de la boucle.
3. Démontrez que

$$(0 \leq i \leq n) \wedge \left( s = \sum_{j=0}^{i-1} (V[j] \times W[j]) \right)$$

est un invariant correct pour la boucle.

4. Servez-vous de ces informations pour démontrer que la boucle fait bien ce que vous aviez prévu (en supposant qu'elle se termine).

**Ex. 10.11.** À quoi sert la boucle suivante ?

```
q = 0
r = j1
while r >= j2:
 q += 1
 r -= j2
```

Démontrez que cette boucle est correcte (qu'elle fait bien ce que vous venez de prédire) en supposant que  $j1$  soit un entier positif et  $j2$  un entier strictement positif. Pour ce faire, vérifiez que  $Inv \equiv (r + q \times j2 = j1) \wedge (q \in \mathbb{N}) \wedge (r \in \mathbb{N})$  est un invariant, et donnez la postcondition du `while`.

**Ex. 10.12.** Voici la fonction *plus* :

```
def plus (x,y):
 while y>0:
 x += 1
 y -= 1
 return x
```

Démontrez que cette boucle calcule bien la somme de  $x$  et  $y$  dans  $x$ . Pour ce faire, montrez que l'invariant du `while` est  $Inv \equiv (y \geq 0) \wedge (x + y = x_0 + y_0)$ , où  $x_0$  et  $y_0$  dénotent les valeurs initiales de  $x$  et  $y$  respectivement, et montrez la terminaison de la fonction. On suppose initialement que  $y \geq 0$ .

**Ex. 10.13.** La boucle suivante est censée imprimer la valeur maximale de  $v$ . Démontrez que  $Inv \equiv \forall j \cdot (0 \leq j \leq i < n) \rightarrow (V[i] \geq V[j])$  est un invariant correct pour la boucle.

```
i = 0
while i < n-1 :
 if v[i] > v[i+1] :
 v[i], v[i+1] = v[i+1], v[i]
 i += 1
print (v[n-1])
```

**Ex. 10.14.\* (Question de janvier 2003)** On se donne la fonction suivante, qui fait l'hypothèse que le vecteur  $V$  passé en paramètre est trié.

```
def f(V,x):
 n = len(V)
 bi = 0
 bs = n - 1
 m = (n - 1) // 2

 while (bs >= bi) and (V[m] != x) :
 if V[m] < x :
 bi = m + 1
 else :
 bs = m - 1
 m = (bi + bs) // 2

 if x != V[m] :
 m = -1

 return m
```

1. Quelle est l'utilité de cette fonction ?
2. Montrez que

$$\left( (\exists i \cdot 0 \leq i < n \wedge V[i] = x) \rightarrow bi \leq i \leq bs \right) \\ \wedge (0 \leq bi < n + 1) \wedge (-1 \leq bs < n) \wedge m = \lfloor \frac{bi+bs}{2} \rfloor$$

est un invariant correct pour la boucle.

3. Démontrez la terminaison de la boucle.

**Ex. 10.15.\* (Examen de janvier 2005)** Soit le code suivant qui suppose que  $n$  est une constante entière strictement positive.

```
def test(V):
 n = len(V)
 delta = 0
 i = 0
 while i < n-1 and V[i] + delta <= V[i+1]:
 delta = V[i+1] - V[i]
 i += 1
 return i == n-1
```

1. Exprimez en français et en logique du premier ordre la condition nécessaire et suffisante sur le vecteur  $V$  pour que la fonction `test` renvoie la valeur *True*.
2. Montrez que

$$\forall j \cdot (0 \leq j < i - 1) \rightarrow (V[j + 1] - V[j] \leq V[j + 2] - V[j + 1])$$

est un invariant correct pour la boucle.

3. Démontrez la terminaison de la boucle.



# Chapitre 11

## Récurtivité

### Exercices préparatoires

Matière à réviser :

- la gestion de mémoire (*stack frames*) ;
- les fonctions.

**Prép. 11.1.** Une fonction `S (n)` qui calcule la somme des  $n$  premiers entiers (i.e.  $S_n = \sum_{i=1}^n i$ ).

**Prép. 11.2.** Une fonction `F (n)` qui calcule la  $n$ -ème valeur de la suite  $F_n$  où  $F_0 = 0$ ,  $F_1 = 1$  et  $F_n = F_{n-2} + F_{n-1}$ .

**Prép. 11.3.** Expliquez les avantages et inconvénients des procédures récursives (c'est-à-dire où on empile des appels récursifs de fonctions) vis-à-vis des procédures itératives (boucles).

### Exercices en séance

Écrivez les fonctions suivantes de manière récursive :

**Ex. 11.1.** Une fonction `ackermann (m, n)` qui implémente la fonction d'Ackermann  $A(m, n)$  définie comme suit :

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ A(m - 1, 1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{si } m > 0 \text{ et } n > 0. \end{cases}$$

Vérifiez que `ackermann (3, 6)` donne 509. Que se passe-t-il pour de plus grandes valeurs ? Pourquoi ?

**Ex. 11.2.** Une fonction `pgcd (x, y)` qui calcule le plus grand commun diviseur de deux entiers positifs  $x$  et  $y$ . Pour cela, utilisez la méthode d'Euclide. Cette méthode se base sur l'observation que si  $r$  est le reste de la division de  $x$  par  $y$ , alors le pgcd de  $x$  et  $y$  est égal au pgcd de  $y$  et  $r$ . Considérez comme cas de base que le pgcd de  $x$  et 0 est  $x$ .

**Ex. 11.3.** Une fonction `factorielle (n)` qui calcule et renvoie la factorielle de  $n$  (noté  $n!$ ). Pour rappel :  $n! = 1 \times 2 \times 3 \dots \times n$ .

**Ex. 11.4.** Une fonction `puissance (x, n)` qui calcule et renvoie la  $n$ -ième ( $n \in \mathbb{N}$ ) puissance de  $x$  sans utiliser l'opérateur `**`.

**Ex. 11.5.** Une fonction `triangle_pascal(i, j)` qui, étant donné deux entiers positifs  $i$  et  $j$ , renvoie la valeur située à la  $i^{\text{e}}$  ligne et la  $j^{\text{e}}$  colonne du triangle de Pascal. Le triangle de Pascal (voir Figure 11.1) est construit comme suit :

- La valeur en  $(0, 0)$  vaut 1.
- Pour toute case du triangle, la valeur de cette case est la somme de la valeur située au-dessus et de celle située au-dessus à gauche.
- Pour toutes les autres « cases », considérez une valeur nulle.

|   | 0 | 1 | 2   | 3 | 4 | 5 |
|---|---|---|-----|---|---|---|
| 0 | 1 |   |     |   |   |   |
| 1 | 1 | 1 |     |   |   |   |
| 2 | 1 | 2 | 1   |   |   |   |
| 3 | 1 | 3 | 3   | 1 |   |   |
| 4 | 1 | 4 | 6   | 4 | 1 |   |
| 5 |   |   | ... |   |   |   |

FIGURE 11.1 – Triangle de Pascal

**Ex. 11.6.** Une fonction `contient(n, d)` qui renvoie `True` si et seulement si l'entier positif représenté par  $n$  contient le chiffre  $d \in \{0, \dots, 9\}$  représenté par  $d$  (astuce : utilisez la division entière et le modulo).

**Ex. 11.7.** Une fonction `inverse(n)` qui renvoie le miroir d'un entier positif  $n$ . Par exemple le miroir de 475 est 574.

**Ex. 11.8.** Une fonction `est_multiple(n, d)` qui renvoie `True` si et seulement si l'entier positif  $n$  est un multiple de l'entier positif  $d$ . Vous ne pouvez pas utiliser les opérateurs `//`, ni `/`, ni `%`.

**Ex. 11.9.** Une fonction `divE(n, d)` qui renvoie la division entière entre  $n$  et  $d$  (i.e.  $n//d$ ) sans utiliser l'opérateur `//`.

**Ex. 11.10.\*** Une fonction `concatenation(L1, L2)` qui renvoie une liste  $L$  - la concatenation de listes  $L1$  et  $L2$ . par exemple : si  $L1 = [1, 2]$  et  $L2 = [3, 4]$ , alors la fonction `concatenation(L1, L2)` doit renvoyer la liste  $[1, 2, 3, 4]$ . Vous ne pouvez pas utiliser l'opérateur `+`, ni la méthode `extend()`.

# Chapitre 12

## Fichiers et exceptions

### Exercices préparatoires

Matière à réviser :

- la fonction `open(file, mode)` avec modes d'ouverture *read*, *write* et *append* ;
- les méthodes `close()`, `write()`, `read()`, `readline()`, `readlines()`, `seek()` sur les objets fichiers ;
- le module `os` et la notion de chemins *absolus* et *relatifs* ;
- les exception plus fréquentes : `ValueError`, `TypeError`, `IndexError`, `IOError` ;
- lever une exception via `raise` ;
- gérer des exceptions via `try-except` ;
- les fonctions `ord()` et `chr()` (voir exercice 4.8, p. 19).

Lisez le *Mode d'emploi introductif pour les salles du NO4 et NO3*, partie « la console Linux ».

**Prép. 12.1.** Écrivez une fonction `lecture(nomFichier)` qui affiche le contenu du fichier.

**Prép. 12.2.** Écrivez une fonction `ecriture(nomFichier, chaine)` qui crée un fichier vide portant le nom `nomFichier` (ou, s'il existe, vide ce fichier) et écrit la chaîne de caractères `chaine` dans celui-ci.

**Prép. 12.3.** Écrivez une fonction `ajouter(nomFichier, chaine)` qui ajoute la chaîne de caractères `chaine` à la fin du fichier dont le nom est la chaîne de caractères `nomFichier`.

**Prép. 12.4.** Les fonctions manipulant des fichiers lèvent, en cas de problème, une exception de type `IOError`. Écrivez un code principal utilisant la fonction écrite à l'exercice préparatoire 12.1 et qui gère correctement cette exception. Ainsi, si on tente d'ouvrir un fichier inexistant en lecture, votre code devra le signaler par un message d'erreur à l'utilisateur plutôt que de faire planter l'interpréteur.

**Prép. 12.5.** Montrez quels seront les résultats du code ci-dessous. Expliquez brièvement ce que doit contenir le fichier lu pour déclencher chacun des cas d'exception.

```
try:
 f = open('myfile.txt')
 s = f.readline()
 i = int(s.strip())
except IOError:
 print("I/O error.")
except ValueError:
```

```

print("Could not convert data to an integer.")
except:
 print("Unexpected error.")
 raise

```

## Exercices en séance

**Ex. 12.1.** Écrivez une fonction `wc(nomFichier)` qui doit ouvrir le fichier en question et renvoyer le nombre de caractères (sans compter les caractères de retour à la ligne), le nombre de mots et le nombre des lignes du fichier. Nous définissons ici un mot comme étant une chaîne de caractères (maximale) répondant *True* à la méthode `isalnum()`.

**Ex. 12.2.** Écrivez une fonction `replace(in_path, out_path, pattern, subst)` qui doit ouvrir le fichier dont le chemin est `in_path` et remplacer dans ce dernier toutes les occurrences du string `pattern` par `subst`. Le résultat de cette modification devra être écrite dans le fichier de chemin `out_path`.

**Ex. 12.3.** À l'aide d'un dictionnaire, écrivez les fonctions suivantes :

- `file_histogram(fileName)` qui prend en paramètre le nom, sous forme d'une chaîne de caractères, d'un fichier texte et qui renvoie un dictionnaire contenant la fréquence absolue (c'est-à-dire le nombre d'occurrences) de chaque lettre dans le texte contenu dans le fichier.
- `vowels_histogram(fileName)` qui prend en paramètre le nom, sous forme d'une chaîne de caractères, d'un fichier texte et qui renvoie un dictionnaire contenant la fréquence absolue des chaque suite de voyelles (par exemple : « i », « e », « oe », « oui », « eui », « you », etc.). Notez que si on trouve « oe » par exemple, il ne faut pas compter une occurrence de « o » ni de « e ».
- `words_by_length(fileName)` qui prend en paramètre le nom, sous forme d'une chaîne de caractères, d'un fichier texte et qui renvoie un dictionnaire associant à une valeur entière  $\ell$ , la liste des mots dans le fichier de longueur  $\ell$ .

**Ex. 12.4.** Reprenez la fonction `trace(M)` écrite à l'exercice 6.1 (p. 26) et ajoutez-y la capture avec une ou plusieurs exceptions des erreurs pouvant survenir dans le cas où la matrice ne serait pas adaptée pour le calcul de trace (c'est-à-dire quand elle n'est pas carrée ou que les éléments de la diagonale ne peuvent être sommés, par exemple si on a un mélange de nombres et de chaînes de caractères).

```

def trace(M):
 dimension = len(M)
 if dimension != len(M[0]):
 res = None
 else:
 res = 0
 for i in range(dimension):
 res += M[i][i]
 return res

```

**Ex. 12.5. Pendu (mini-projet)** Nous vous demandons de développer un jeu du pendu. Dans ce dernier, le joueur essaie de deviner en un nombre limité de tentatives, lettre par lettre, un mot choisi par l'ordinateur. Ce dernier choisira au hasard un des mots contenu dans le fichier `dictionary.txt`. Le joueur n'a droit qu'à dix tentatives infructueuses (à la 10<sup>e</sup> erreur, le joueur a perdu). À chaque étape, un récapitulatif de l'état du jeu (c'est-à-dire le nombre restant

de tentatives infructueuses, les lettres déjà jouées, ainsi que le mot à deviner dans lequel chaque lettre encore non trouvée est représentée par une astérisque) doit être affiché avant le choix suivant de lettre du joueur.

À chaque fois que le joueur entrera une lettre, le programme vérifiera si ce dernier est dans le mot caché. Si c'est le cas, il ajoute les nouvelles lettres dévoilées, sinon le nombre de tentatives restantes est décrémenté. Si le nombre de tentatives restantes atteint 0, le joueur a perdu. Si l'entière du mot est dévoilé, la partie est gagnée et le score est affiché. Ce dernier est calculé à l'aide de la formule suivante :

$$\text{score} = \ell + r$$

où  $\ell$  est la longueur du mot et  $r$  le nombre de tentatives restantes. Pour le choix au hasard du mot, nous vous demandons d'utiliser la bibliothèque `random` et `randint()` plus particulièrement.

**Ex. 12.6. Checksum (mini-projet)** Les données peuvent parfois être perdues ou corrompues pendant leur transmission. On vous demande d'implémenter un *code détecteur d'erreur* et un *code correcteur d'erreur*. Ces codes permettent de détecter et de corriger les erreurs éventuelles de transmission.

Dans ce mini-projet supposez que le fichier texte ne contient que des caractères 0 et des caractères 1 et que toutes les lignes ont la même longueur.

**Code détecteur** L'idée de ce code est d'ajouter des informations redondantes dans le message. Nous allons diviser le fichier en blocs et ajouter un bit à chaque bloc pour que le nombre de bits à 1 soit impair. Tous les nouveaux bits (les bits de parité) qu'on calcule ainsi constituent un *checksum*.

1. Écrivez une fonction `addChecksum(filename)` qui lit le contenu d'un fichier, calcule le checksum et l'ajoute à la fin du fichier. Prenez donc une ligne pour un bloc, voir exemple 12.1.
2. Écrivez une fonction `verifyChecksum(filename)` qui lit un fichier avec le checksum et vérifie si le checksum est correct (renvoie `True` si c'est le cas).

File :

```
1 0 1 1
0 0 0 1
1 0 1 0
```

Les bits de parité qu'il faut ajouter : 0 (ligne 1), 0 (ligne 2), 1 (ligne 3).

Après l'exécution de `addChecksum` :

```
1 0 1 1
0 0 0 1
1 0 1 0
0 0 1
```

FIGURE 12.1 – Exemple `addChecksum(filename)`.

**Code correcteur** L'idée de ce code est de modifier le code détecteur, pour pouvoir aussi corriger des erreurs. On va donc ajouter les bits de parité pour chaque ligne et pour chaque colonne, voir l'exemple Figure 12.2 . Ce code permet de corriger au plus une erreur.

1. Ecrivez une fonction *addChecksumCorr(filename)* qui lit le contenu d'un fichier, calcule le checksum et l'ajoute à la fin de fichier.
2. Ecrivez une fonction *verifyAndCorrect(filename)* qui lit un fichier avec le checksum, vérifie si le checksum est correcte et corrige le fichier si le fichier est corrompu.

File :

```
1 0 0 1
0 0 0 1
1 0 1 0
```

Calcule de bits de parité :

```
1 0 0 1 1
0 0 0 1 0
1 0 1 0 1
```

```
1 1 0 1
```

Fichier apres l'execution de addChecksumCorr :

```
1 0 1 1
0 0 0 1
1 0 1 0
1 0 1
1 1 0 1
```

FIGURE 12.2 – Exemple addChecksum(filename).

**Ex. 12.7. \*\*** Faites l'exercice 12.6 sans supposer que le fichier contient que des 1 et des 0 et sans supposer que tous les lignes ont la même longueur.

# Chapitre 13

## Introduction aux classes

### Exercices préparatoires

Matière à réviser :

- notions de classe, instance, attribut,
- différence entre fonction pure et méthode d'instance,
- méthode : `__init__`, `__str__`, `__repr__`,
- surcharge d'opérateur.

**Prép. 13.1.** Que fait le code suivant ? Expliquez.

```
class A:
 def __init__(self, msg="bright"):
 self.attr = msg

 def __repr__(self):
 return self.attr

 def set(self, msg):
 self.attr = msg

x = A()
print(x)
y = A("side of life")
print(y)
y.set(x.attr)
print(y)
```

**Prép. 13.2.** Écrivez une classe `Point` qui va définir un point dans un espace de dimension 3. Nous vous demandons de définir un constructeur avec valeurs par défaut, de définir la méthode `__repr__(self)` de la classe pour pouvoir utiliser `print()` sur vos objets `Point` et de définir la méthode `distance(self, point)` qui prend un objet de type `Point` en entrée et qui renvoie la distance entre les deux points.

### Exercices en séance

**Ex. 13.1.** Écrivez une classe `StopWatch` qui modélise un chronomètre. Cette classe doit contenir les méthodes suivantes :

- `start` qui lance le chronomètre.

- `stop` qui arrête le chronomètre.
- `getElapsedTime` qui renvoie le temps écoulé en secondes depuis le dernier lancement du chronomètre (s'il est en train de tourner), ou le temps écoulé entre le dernier départ et le dernier arrêt si le chronomètre a été arrêté. Si le chronomètre n'a jamais été lancé, cette méthode devra renvoyer 0.

**Conseil :** utilisez le module `time` fourni par Python, en particulier la fonction `time.time()`.

**Ex. 13.2.** Écrivez une classe `Monome`. Un monôme est un polynôme composé d'un seul terme. Par exemple,  $3x^4$  ou  $7x^5$  ou encore 8 sont des monômes. Un monôme est caractérisé par deux attributs : son coefficient et son degré. Votre classe devra donc comporter deux attributs, une méthode d'initialisation avec valeurs par défaut ainsi que deux méthodes d'accès (*getters*) qui renvoient les valeurs des attributs. Enfin, la méthode `__repr__` est demandée.

**Ex. 13.3.** Définissez les méthodes `__add__`, `__sub__`, `__mul__`, `__floatdiv__`, `__truediv__`, `__lt__`, `__gt__`, `__eq__` pour la classe `Monome` définie précédemment en respectant les contraintes suivantes :

- le méthode `__add__`, `__sub__`, `__mul__`, `__truediv__`, `__floatdiv__`, doivent renvoyer un monôme ;
- si les monômes ont des degrés différents, les méthodes `__add__` et `__sub__`, doivent lever une exception ;
- nous souhaitons que la relation d'ordre entre deux monômes soit la suivante :  $a^x \langle op \rangle b^y \iff x \langle op \rangle y$ , où  $\langle op \rangle$  est un opérateur de l'ensemble  $\{<, \leq, =, \geq, >\}$ .

**Exemple :**  $2x^5 > 3x^4$  est vrai .

**Ex. 13.4.** Écrivez une classe `Polynome` qui va définir un polynôme. Un polynôme est une somme finie de monômes ; nous pouvons donc représenter un polynôme par une liste de monômes. Nous vous demandons d'écrire les méthodes `__init__`, `__str__`. La méthode `__str__(self)` devra formater l'affichage du polynôme en respectant l'ordre des monômes tel que donné à l'exercice 13.3 :

```
>>> p = Polynome([Monome(-2,4), Monome(3,5), Monome(9,0), Monome(-11,1)])
>>> print(p)
Polynome : 3 x^5 - 2 x^4 - 11 x + 9
```

**Ex. 13.5.** Ajoutez à la classe `Polynome` la méthode `reduce` qui va faire l'addition entre monômes du même degré.

**Exemple :**  $3x^5 + 2x^4 + 3x^4 + 2x^2 + 11x + 3x + 9 + 10$  devient  $3x^5 + 5x^4 + 2x^2 + 14x + 19$ .

**Ex. 13.6.\*** Définissez les méthodes `__add__` et `__mult__` sur la classe `Polynome` qui permettent de faire l'addition et la multiplication entre deux polynômes.

```
>>> p1 = Polynome([Monome(2,4), Monome(3,5), Monome(9,0)])
>>> p2 = Polynome([Monome(2,5), Monome(4,0), Monome(10,1)])
>>> p3 = p1 + p2
>>> p4 = p1 * p2
>>> print(p3)
Polynome : 5 x^5 + 2 x^4 + 10 x + 13
>>> print(p4)
Polynome : 6 x^10 + 4 x^9 + 30 x^6 + 50 x^5 + 8 x^4 + 90 x + 36
```



**Ex. 13.7.** Écrivez une classe `Rationnel` qui va définir un nombre rationnel. Un nombre rationnel est un nombre qui peut s'exprimer comme le quotient de deux entiers  $\frac{a}{b}$ , où  $a$  et  $b$  sont deux entiers (avec  $b$  non nul). On appelle  $a$  le numérateur et  $b$  le dénominateur. Votre classe devra avoir deux champs attributs et définir les méthodes suivantes : (`__init__`, `__add__`, `__sub__`, `__mul__`, `__floatdiv__`, `__truediv__`, `__str__`, `__float__`).

**Ex. 13.8.\*\*** Ajoutez à la classe `Polynome` la méthode `composition` qui va faire la composition entre deux polynômes.

**Exemple :** Si pour tout réel  $x$  :

$$\begin{aligned} f(x) &= -3x^2 + 4x - 2 \\ g(x) &= 4x - 1 \end{aligned}$$

alors le polynôme  $f \circ g$  est le polynôme défini pour tout réel  $x$  par :

$$\begin{aligned} (f \circ g)(x) &= f(g(x)) \\ &= -3.[g(x)]^2 + 4.[g(x)] - 2 \\ &= -3.[4x - 1]^2 + 4.[4x - 1] - 2 \\ &= -3.[16x^2 - 8x + 1] + 4.[4x - 1] - 2 \\ &= -48x^2 + 24x - 3 + 16x - 4 - 2 \\ &= -48x^2 + 40x - 9 \end{aligned}$$

Par contre,  $g \circ f$  est le polynôme défini pour tout réel  $x$  par :

$$\begin{aligned} (g \circ f)(x) &= g(f(x)) \\ &= 4.[f(x)] - 1 \\ &= 4.[-3x^2 + 4x - 2] - 1 \\ &= -12x^2 + 16x - 8 - 1 \\ &= -12x^2 + 16x - 9 \end{aligned}$$

# Chapitre 14

## Base de données

### Exercices préparatoires

**Prép. 14.1.** Les bases de données sont utilisées pour stocker un grand nombre de données de manière structurée et optimisée afin de pouvoir accéder rapidement et facilement à une information. Dans une base de données, nous retrouvons plusieurs tables (que nous pouvons considérer comme différents ensembles contenant des informations différentes : des contacts, des véhicules, des adresses, etc.). Chacune contient plusieurs lignes (entrées); chacune d'entre elles correspond à une entité (une personne, un événement ou un véhicule par exemple).

Dans une table, une entrée est définie par une clé. Celle-ci représente l'entrée de manière unique dans la table : une clé ne peut pas correspondre à deux ou plus d'entrées dans une table. Nous allons représenter une table d'une base de données à l'aide d'un dictionnaire et de tuples.

Pour cet exercice, nous allons considérer une table d'étudiants contenant leur nom, âge et taille, comme par exemple à la Table 14.1. Le nom de l'étudiant servira de clé dans la table. Nous ne pourrions donc pas avoir deux entrées dans la table pour un même étudiant.

|           |               |            |               |
|-----------|---------------|------------|---------------|
| Etudiants |               |            |               |
| clé       | <b>nom</b>    | <i>age</i> | <i>taille</i> |
|           | .             | .          | .             |
|           | .             | .          | .             |
|           | .             | .          | .             |
|           | <b>Arnaud</b> | 35         | 1.78          |
|           | .             | .          | .             |
|           | .             | .          | .             |
|           | .             | .          | .             |

TABLE 14.1 – Extrait d'une table pour stocker des étudiants dans une base de données.

Nous vous demandons d'écrire la fonction de remplissage `ajouter(table)` qui doit demander les données à l'utilisateur jusqu'à ce que ce dernier entre `none` au clavier et rajouter les informations dans la table `table` donnée en paramètre.

Vous devez ensuite écrire la fonction de consultation `consulter(table)` qui doit elle aussi demander à l'utilisateur de rentrer le nom d'un étudiant présent dans la table `table` donnée

en paramètre et dont on veut consulter les informations (et ce jusqu'à ce que l'utilisateur rentre `none` au clavier). La fonction doit afficher ses informations de la manière suivante :

```
"Nom : Jeannot Lapin - age : 23 ans - taille : 1.30m"
```

## Exercices en séance

Nous allons pour cette Super Séance perfectionner notre base de données commencée à l'exercice 14.1.

**Ex. 14.1.** Reprenez la fonction `ajouter(table)` écrite pour l'Ex. 14.1 sur les dictionnaires et écrivez comment capturer une exception dans le cas où un des paramètres passés par l'utilisateur ne sont pas du bon type. Dans le cas où l'exception est levée, vous afficherez un message d'erreur décrivant.

```
def ajouter(table):
 rep = input("Nom ? ")
 while rep != None:
 if rep not in table:
 age = input("Âge ? ")
 taille = input("Taille ? ")
 table[rep] = (age, taille)
 else:
 print("Erreur : l'étudiant est déjà encodé.")
 rep = raw_input("Nom ? (none pour arrêter) ")
 return table
```

**Ex. 14.2.** Écrivez une fonction `conversion(table, critere)` dont l'objectif est de changer le critère utilisé comme clé dans une table. Comme à l'Ex. 14.1, chaque entrée de la table fournie à la fonction est composée d'une clé et d'un tuple comprenant les valeurs des autres critères. La variable `critere` est l'indice, dans le tuple, de la clé du nouveau dictionnaire. Par exemple, si nous reprenons l'Ex. 14.1, l'objectif de la fonction, si on lui passe 0 comme critère, serait de construire un nouveau dictionnaire tel que la clef est l'âge de l'étudiant, tandis que le nom (qui était la clef du dictionnaire d'origine) remplacerait la valeur de la colonne « âge ». Si le nouveau critère n'est pas unique pour les différentes entrées de la table, les nouveaux tuples seront insérés dans des listes. Vous considérerez que le nouveau critère est toujours utilisable comme clé (on peut donc supposer que ce n'est jamais une liste ou un dictionnaire par exemple). Voici un exemple de conversion :

```
>>> t = {'Kate' : (35, 1.90), 'George' : (35, 1.99), 'Jamie' : (23, 1.73)}

>>> print (convert(t,0))
{35: [('Kate', 1.90), ('George', 1.99)], 23: [('Jamie', 1.73)]}

>>> print (convert(t,1))
{1.90: [(35, 'Kate')], 1.73: [(23, 'Jamie')], 1.99: [(35, 'George')]}
```

**Ex. 14.3.** L'exercice 14.1 n'est pas très pratique car, après avoir exécuté le code et fermé l'interpréteur, le contenu de la table n'existe plus au prochain lancement de l'interpréteur. Pour maintenir les données persistantes d'une exécution à l'autre, nous allons donc les sauvegarder dans un fichier.

Nous vous demandons d'écrire deux fonctions : `sauvegarder(table, fichier)` qui sauvegarde une table dans un fichier et `charger(fichier)` qui renvoie une table remplie à partir d'un fichier (nous gardons la même conception de table que pour l'exercice 14.1).

Chaque ligne du fichier texte correspondra à un élément du dictionnaire (de la table) que l'on désire sauvegarder. Elle sera formatée de manière à bien séparer les informations. Nous respecterons le format suivant :

```
nom_de_l'etudiant@age#taille
```

Voici un exemple :

```
Jerome@24#1.74
Naim@25#1.76
...
```

Pour ouvrir un fichier en écriture, il faut ajouter un paramètre supplémentaire à la fonction `open`. Il nous faut donc utiliser l'instruction :

```
fichier = open('nom_du_fichier', 'w')
```

où `'w'` permet d'indiquer que nous ouvrons le fichier en écriture (*write*).

**Ex. 14.4.** Stocker de l'information est une chose pratique, mais pouvoir la modifier en est une autre chose tout aussi importante et utile.

Une table contient plusieurs entrées, il faut donc toujours préciser quelle ligne on désire modifier. Nous supposons ici que la clé d'une entrée ne peut jamais être modifiée.

La fonction `update(table)` reçoit en paramètre une table contenant des étudiants et qui demandera le nom de l'étudiant dont on veut modifier les informations. Si l'étudiant n'existe pas, la fonction lèvera une exception (via `raise` par exemple) de type `KeyError` qui servira à afficher un message d'erreur plus tard.

La fonction devra alors parcourir les champs de l'entrée indiquée en demandant si l'on désire en modifier la valeur :

```
> Veuillez fournir le nom de l'étudiant à modifier :
> Jerome
> L'age de Jerome est : 24 (entrez une nouvelle valeur ou \
 entrez 0 pour ne pas modifier cette valeur)
> 0
> La taille de Jerome est : 1.74 (entrez une nouvelle valeur ou \
 entrez 0 pour ne pas modifier cette valeur)
> 1.77
```

**Ex. 14.5.** Il peut être aussi utile de pouvoir supprimer une entrée d'une table. À nouveau, il nous faut alors préciser la table et l'entrée que nous désirons y retirer.

Écrivez une fonction `delete(table)` qui demande le nom d'un l'étudiant et qui retire l'entrée correspondante dans la table. Si la clé indiquée n'existe pas, la fonction lèvera une exception de type `KeyError`.

Pour vous aider, utilisez la méthode `pop()` sur les dictionnaires qui prend une clef en paramètre et qui supprime l'entrée correspondante dans le dictionnaire désigné.

**Ex. 14.6.** Nous vous demandons pour finir de rédiger un programme « chef d'orchestre » qui nous permet de créer, enregistrer et modifier une table dans une base de données à l'aide des fonctions définies précédemment. Il est demandé que le programme charge automatiquement la table sauvegardée au démarrage. Veillez à ce que la table soit sauvegardée aux bons moments de sorte que si nous quittons le programme, nous retrouvons bien toutes les données de la table.

```
(c)onsulter la table
(a)jouter une entrée
(m)ettre à jour une entrée
(s)upprimer une entrée
(q)uitter
SGBD>> s
Quel est le nom de l'étudiant à supprimer ?
Yami
L'étudiant recherché n'existe pas dans la table.
(c)onsulter la table
(a)jouter une entrée
(m)ettre à jour une entrée
(s)upprimer une entrée
(q)uitter
SGBD>> ...
```

# Chapitre 15

## Révision

### Exercices préparatoires

**Prép. 15.1.** Revoir toute la matière :-)

### Exercices en séance

**Ex. 15.1.** Écrivez une fonction `multiple9` qui prend un paramètre entier  $n \in \{1, \dots, 9\}$  et qui renvoie le produit de  $n$  par 9 sous forme d'une chaîne de caractères, sans utiliser l'opérateur de multiplication ni de boucles.

**Ex. 15.2.** Ecrivez une fonction `compress` correspondante à l'exercice 5.13

**Ex. 15.3.** Écrivez une fonction `deepprint(dico, depth)`, qui va afficher les éléments d'un dictionnaire en précisant pour chaque élément du dictionnaire la clé et sa valeur. Si la valeur est elle-même un dictionnaire, il faudra effectuer récursivement son affichage en indentant l'affichage ; le paramètre `depth` est en entier qui permet de préciser à quel niveau d'indentation on se situe (0 signifie pas d'indentation, 1 signifie qu'on a un niveau d'indentation, etc.). Voici un exemple d'affichage :

```
Nom: Pierre-Paul
Prénom: Jacques
Âge: 26
Cours:
 INFO-F-101:
 NoteFinale: 14
 InterroJanvier: 12
 Projets: 10
 INFO-F-102: 17
Matricule: 372910
```

**Ex. 15.4.** On stocke dans un fichier le nom et l'adresse de différentes personnes (une personne et son adresse par ligne). Par exemple :

```
David Roman#Rue des chateaux, 123#1100 Youplala#Belgique
Eric Degeers#Avenue de l'avenue, 13#11800 Mouahahah#France
John Meersman#Route de la vie, 55#1000 Forest#Belgique
...
```

Vous devez réaliser un programme qui permettra à l'utilisateur d'afficher ces informations selon différents tris :

- En fonction du nom, puis du code postal, puis du pays
- En fonction du pays, puis de la rue, puis du nom
- En fonction du code postal, puis du nom

Le programme devra continuellement demander à l'utilisateur comment il veut afficher les informations jusqu'à ce que ce dernier demande de stopper le programme.

**Ex. 15.5.** On vous demande d'implémenter la fonction `tassement` qui prends en argument une liste `L` (qui contient des nombres naturels). La fonction doit modifier la liste `L` pour que tous les éléments 0 se retrouvent à la fin de la liste. On vous demande d'implémenter cette fonction de deux façons (voir les exemples) :

- L'ordre de tous les éléments différents de 0 peut être modifié (`tassementNonOrd`)
- L'ordre de tous les éléments différents de 0 ne doit pas changer (`tassementOrd`),

Exemple :

```
>>> L = [1, 4, 2, 0, 4, 0, 3, 2, 0, 0, 1]
>>> tassementOrd(L)
>>> print(L)
>>> [1, 4, 2, 4, 3, 2, 1, 0, 0, 0, 0]
>>> L = [1, 4, 2, 0, 4, 0, 3, 2, 0, 0, 1]
>>> tassementNonOrd(L)
>>> print(L)
>>> [1, 4, 2, 1, 4, 2, 3, 0, 0, 0, 0]
```

**Ex. 15.6.** Écrivez une fonction `appartient` qui prend deux arguments, le premier est une chaîne de caractères, le deuxième est une liste de chaînes de caractères. La fonction vérifie l'appartenance de chaque élément de la liste dans le premier argument. Si le test est vérifié l'élément est ajouté à une première liste, sinon il est ajouté dans une autre liste. La fonction renvoie ensuite les deux listes créées. (Note : la fonction ne peut pas faire appel à la fonction `find`)

**Ex. 15.7. (Examen Septembre 2006, Info-f-206)**

Un *nombre parfait* est un nombre entier naturel qui est égal à la somme de ses diviseurs, y compris 1, mais excepté lui-même. On vous demande d'écrire une fonction `nombre_parfait(nbr)` qui renvoie la valeur booléenne `true` si `nbr` est un nombre parfait et la valeur `false` sinon.

Exemples de nombres parfaits :

```
6 = 1 + 2 + 3\\
28 = 1 + 2 + 4 + 7 + 14\\
496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248
```

**Ex. 15.8. (Examen bidon 2012, Info-f-206)**

On considère des chaînes de caractères dans l'alphabet  $\{A, C, G, T\}$ , représentant des séquences ADN. Une telle chaîne est dite *complémentaire* d'une autre, si on obtient la deuxième en remplaçant chaque occurrence de la lettre *A* par *T*, *T* par *A*, *G* par *C*, et *C* par *G* dans la première. Par exemple, les séquences *ACCGAT* et *TGGCTA* sont complémentaires.

Ecrire une fonction qui reçoit en paramètre une liste de telles chaînes, et renvoie une nouvelle liste dans laquelle les paires complémentaires sont éliminées. Pour chaque paire complémentaire présente dans la liste de départ, seule la seconde chaîne de la paire sera conservée dans la liste résultat. Par exemple, la liste suivante :

```
['ATCC', 'ACGG', 'TGCC', 'ACC', 'TAGG']
```

contient deux paires complémentaires : 'ACGG' et 'TGCC' d'une part, et 'ATCC' et 'TAGG' d'autre part. On doit éliminer la première chaîne (dans l'ordre de la liste) de chaque paire. La liste renvoyée par la fonction sera donc :

```
['TGCC', 'ACC', 'TAGG']
```

On supposera que toutes les chaînes de la liste de départ sont distinctes.

### Ex. 15.9.\* (Examen Janvier 2013)

On vous demande d'implémenter la fonction `alphaCount()` qui prend en argument une chaîne de caractères `text`. Vous devez implémenter la fonction `alphaCount()` de deux façons différentes :

1. vous avez le droit d'utiliser tous les outils fournis par *Python*,
2. vous ne pouvez pas utiliser :
  - l'opérateur `in`,
  - les fonctions `set()`, `list()`, `str()`,
  - les méthodes sur des strings, sur des sets ou sur des listes.

La fonction `alphaCount()` doit renvoyer un entier – le nombre de lettres différentes utilisées dans la chaîne `text`. La fonction `alphaCount()` ne doit pas compter les espaces, les nombres, ni les symboles de ponctuation. Les majuscules et les minuscules sont considérées comme la même lettre (e.g. les symboles "b" et "B" représentent la même lettre). Voir les exemples sur la Figure 15.1.

```
>>> alphaCount("bonjour")
>>> 6
>>> alphaCount("acacia")
>>> 3
>>> alphaCount("AaAaAa")
>>> 1
>>> alphaCount("Buvez de ce whisky que le patron juge fameux.")
>>> 26
>>> alphaCount("Monsieur Jack, vous dactylographiez bien mieux que votre ami Wolf!")
>>> 26
>>> alphaCount("Python 3 : alphaCount()")
>>> 10
```

FIGURE 15.1 – Exemple : les appels de la fonction `alphaCount()`.

Notez qu'il est possible d'implémenter la fonction `alphaCount()` avec une complexité en  $O(\text{len}(\text{text}))$ .

Vous pouvez supposer que tous les caractères sont en ASCII et que `text` ne contient pas de caractères accentués.

### Ex. 15.10.\* (Examen Janvier 2012)

La notation polonaise est aussi connue sous le nom de *notation préfixée*. L'idée de cette notation est simple : on écrit d'abord l'opérateur et ensuite les deux opérandes (par exemple « + 2 3 »), par opposition à la notation classique dite *infixe* où l'opérateur est placé entre les deux opérandes (par exemple « 2 + 3 »), voir Figure 15.2.

On vous demande d'écrire une fonction `evaluate` qui prend en paramètre une liste `expression` et qui renvoie un entier `result`, résultat de l'évaluation de l'expression représentée par `expression`. La liste `expression` est structurée de la façon suivante :



- chaque élément est un *string* ;
- chaque élément représente soit un entier, soit un opérateur ;
- les éléments se trouvent dans la liste suivant l'ordre de la notation polonaise (en d'autres termes, nous supposons que la liste est une expression valide).

Voici la liste des opérations possibles avec leurs représentations dans la liste :

- "+" : somme
- "-" : différence
- "\*" : produit
- "/" : division entière
- "%" : reste de la division

```
Expression :
12 + 3 - 4 * 2
Notation polonaise correspondante :
- + 12 3 * 4 2
Représentation sous forme de liste :
["-", "+", "12", "3", "*", "4", "2"]
Résultat: 7
```

FIGURE 15.2 – Exemple : notation polonaise, représentation de l'expression sous forme d'une liste.

Vous pouvez supposer que la liste est correctement structurée, c'est-à-dire que l'ordre et le nombre d'éléments sont corrects. Votre fonction devra toutefois signaler par le biais d'une exception l'éventualité d'une division par zéro.

Essayez d'utiliser le moins de `if-else` possible.

Vous ne pouvez pas utiliser la fonction `eval()` de Python.

Vous pouvez définir d'autres fonctions et écrire du code en plus pour simplifier le code de la fonction `evaluate()`.

Voici encore quelques exemples de liste `expression` :

```
Expression:
1 + 1 + 1 + 1 + 1
Représentation sous forme d'une liste:
["+", "+", "+", "+", "1", "1", "1", "1", "1"]
Résultat: 5

Expression:
((5 - 3) + (1 % 2)) * 8
Représentation sous forme d'une liste:
["*", "+", "-", "5", "3", "%", "1", "2", "8"]
Résultat: 24
```

FIGURE 15.3 – Exemples.

#### Ex. 15.11.\* (Examen Janvier 2013)

Le problème de partition consiste à découper un ensemble d'entiers  $L$  en deux partitions  $L_1$  et  $L_2$  tel que la somme des éléments du premier ensemble soit égale à la somme des éléments du second.

Par exemple, supposons que  $L$  vaut  $[10, 4, 9, 5, 3, 7]$ . Dans ce cas, une solution au problème de partition appliqué à  $L$  est  $L_1$  qui vaut  $[10, 9]$  et  $L_2$  qui vaut  $[4, 5, 3, 7]$ . En effet, la somme des éléments de  $L_1$  est la même que la somme des éléments de  $L_2$ . Plus précisément la somme est égale à 19 dans les deux partitions.

On vous demande d'écrire une fonction partition qui résout le problème de partition. Plus précisément, votre fonction prendra en entrée une liste d'entiers  $L$  et renverra deux listes  $L_1$  et  $L_2$  à l'aide d'un tuple.  $L_1$  et  $L_2$  formeront une partition de  $L$ . De plus, la somme des éléments de  $L_1$  devra être égale à la somme des éléments de  $L_2$ .

Notez que chaque ensemble d'éléments  $L$  n'a pas forcément de solution. Par exemple la liste  $[4, 5]$  n'a pas de solution. Nous supposerons que la liste  $L$  donnée en entrée de votre fonction contient une solution.

## Annexe A

# Swampy Turtle

### Exercices préparatoires

Matière à réviser :

- géométrie euclidienne de base ;
- angles supplémentaires, complémentaires ;
- périmètre d'un polygône, circonférence d'un cercle ;
- somme des angles dans un polygône régulier ;
- angles d'un polygône régulier.

### Exercices en séance

Pour cette séance, vous devez d'abord télécharger *Swampy* pour Python 3 sur le site :

<http://swampy.googlecode.com/files/swampy.1.4.python3.zip>

Une fois le dossier décompressé, créez-y un fichier `<filename>.py`. Pour commencer, tapez le code suivant dans le fichier que vous venez de créer :

```
import turtle
```

```
turtle.reset()
```

Ce morceau de code ne fait que créer un monde et une tortue. Dans Turtle, chaque tortue peut avancer (`forward`), tourner à gauche (`left`) ou tourner à droite (`right`). De plus, sous chaque tortue se trouve un crayon permettant de tracer un trait si le crayon est baissé. La commande `up` permet de lever le crayon alors que la commande `down` permet de le baisser. Les fonctions `left` et `right` prennent un paramètre qui représente l'angle de rotation : `turtle.left(45)` fera tourner la tortue de 45 degrés vers la gauche.

Pour tracer un angle droit, il faut ajouter (après avoir créé la tortue `turtle`) le code suivant :

```
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
```

Vérifiez que ces commandes dessinent bien un angle droit.

**Ex. A.1.** Modifier les commandes de sorte à dessiner un carré.

**Ex. A.2.** Écrivez une fonction `square` qui prend en paramètre une tortue `myTurtle` et qui lui fait dessiner un carré. Appelez ensuite cette fonction en lui passant la tortue `turtle` en paramètre.

**Ex. A.3.** Modifiez la fonction `square` en lui ajoutant un paramètre `length` qui représentera la longueur des côtés du carré. Testez votre fonction avec différentes longueurs.

**Ex. A.4.** Faites une copie de la fonction `square` et nommez-la `polygon`. Cette fonction prendra un paramètre supplémentaire `n` et devra dessiner un polygone régulier à `n` côtés.

**Ex. A.5.** Écrivez une fonction `circle` qui prend en paramètres une tortue `myTurtle` et un rayon `r`. Cette fonction dessinera un cercle de manière approximative à l'aide de la fonction `polygon` (astuce : faites en sorte que  $\text{length} \times n = \text{circonférence}$ ).

**Ex. A.6.** Écrivez une version plus spécifique de `circle`, appelée `arc`, qui prend un paramètre supplémentaire `angle` exprimé en degrés et déterminant la portion de cercle à tracer.

**Ex. A.7. \* (Examen de juin 2011)** Il pourrait être utile, pour certaines applications, de pouvoir utiliser la tortue en précisant plutôt une liste de points dans le plan par laquelle elle doit passer. Notamment, une telle fonctionnalité permettrait de plus aisément utiliser Turtle pour dessiner des figures mathématiques décrites plus facilement sous forme d'équation polaire (reliant un rayon à un angle) que sous forme cartésienne (reliant une abscisse à une ordonnée), tels des cercles ou des spirales.

Nous vous demandons en premier lieu d'écrire une fonction `relier_points()` qui prendra en paramètres une tortue `t` ainsi qu'un nombre arbitraire de tuples de deux entiers  $(x_1, y_1)$ ,  $(x_2, y_2)$ , ...,  $(x_n, y_n)$  représentant des points dans le plan. L'idée de cette fonction sera de considérer que la position courante de la tortue est l'origine  $(0, 0)$  et d'ensuite la faire passer dans l'ordre par tous les points  $(x_1, y_1)$ ,  $(x_2, y_2)$ , etc. Pour ce faire, l'algorithme suivant peut être utilisé :

1. Soit  $i = 0$
2. Considérer le point  $(x_i, y_i)$  dans la liste  $L$  avec  $x_0 = y_0 = 0$
3. Calculer  $(\Delta x, \Delta y) = (x_{i+1} - x_i, y_{i+1} - y_i)$
4. Calculer  $d = \sqrt{(\Delta x)^2 + (\Delta y)^2}$
5. Si  $\Delta x > 0$ , calculer  $\alpha = \arctg(\frac{\Delta y}{\Delta x})$ . Si  $\Delta x < 0$ , calculer  $\alpha = \arctg(\frac{\Delta y}{\Delta x}) + \pi$ . Sinon (si  $\Delta x = 0$ ), il y a trois autres cas à considérer : si  $\Delta y = 0$ , alors  $\alpha = 0$  ; si  $\Delta y > 0$ , alors  $\alpha = \frac{\pi}{2}$  ; sinon  $\alpha = -\frac{\pi}{2}$ .
6. Faire tourner la tortue de  $\frac{360\alpha}{2\pi}$  degrés vers la gauche (`left`)
7. Avancer la tortue d'une distance  $d$  (`forward`)
8. Faire tourner la tortue de  $\frac{360\alpha}{2\pi}$  degrés vers la droite (`rt`)
9. Incrémenter  $i$  et recommencer à l'étape 2 si  $i \leq n$

Dans un second temps, nous vous demandons d'implémenter une fonction `spirale()` d'un seul paramètre  $k$  et qui renverra une liste de points permettant d'approximer une *spirale d'Archimède*. Celles-ci se caractérisent par une équation en coordonnées polaires de la forme  $\rho(\theta) = k\theta$  où  $\rho$  est le rayon,  $\theta$  est l'angle et  $k$  est un nombre réel strictement positif qui caractérise la spirale. Une propriété de ces objets mathématiques est d'avoir une distance partout égale à  $2k\pi$  entre les bras de la spirale (formellement,  $\rho(\theta + 2\pi) = \rho(\theta) + 2k\pi$ ). Pour obtenir une liste de points (partielle) à partir d'une équation quelconque en coordonnées polaires  $\rho(\theta)$ , l'algorithme suivant peut être utilisé :

1. Soit  $\theta = 0$
2. Calculer  $d = \rho(\theta)$

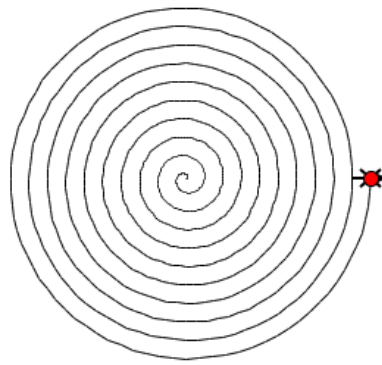


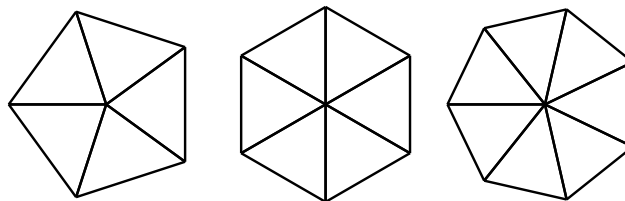
FIGURE A.1 – Spirale d’Archimède ( $k = 2$ ) avec Turtle.

3. Calculer le point  $(x, y)$  correspondant par les relations  $x = d \cos \theta$  et  $y = d \sin \theta$
4. Augmenter  $\theta$  de  $\frac{\pi}{42}$  et recommencer à l’étape 2 jusqu’à ce que  $\theta > 20\pi$ .

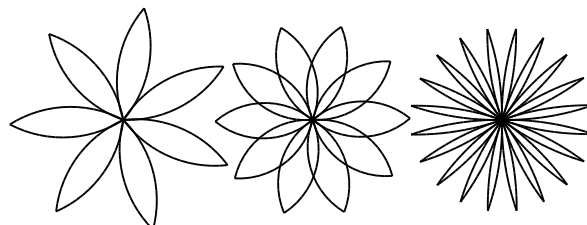
Nous vous demandons enfin d’écrire une fonction `dessin_spirale()` ayant deux paramètres (une tortue  $t$  et le paramètre  $k$  de la spirale) et qui utilise les deux fonctions précédentes pour dessiner une spirale d’Archimède de paramètre  $k$ . La Figure A.1 montre à quoi ressemble le dessin d’une spirale d’Archimède de paramètre  $k = 2$ .

Pour vous assister, vous pouvez bien entendu faire appel aux objets adéquats du module `math` de Python, à savoir les fonctions `cos`, `sin` et `atan` (qui implémente `arctg`) ainsi que la constante `pi`. N’oubliez pas que les fonctions trigonométriques travaillent en radians et non en degrés.

**Ex. A.8. \*\*** Écrivez une fonction permettant de dessiner des figures telles :



**Ex. A.9. \*\*** Écrivez une fonction permettant de dessiner des figures telles :



## Annexe B

# Python Imaging Library

### Exercices préparatoires

Matière à réviser :

- les commandes en terminal (`ls`, `cd`, `mkdir`, `cp`, `rm`, `rmdir` et autres).
- *Mode d'emploi introductif pour les salles du NO4 et NO3*, partie « console Linux ».

La bibliothèque PIL ne supporte pas encore Python 3, on travaillera donc avec la version 2 de Python : lancez `python2` plutôt que `python3` dans le terminal. Notez qu'en Python 2, la syntaxe de la fonction `print()` est légèrement différente : on écrit par exemple `print "Hello!"` au lieu de `print("Hello!")`. Une autre différence est le mode de fonctionnement de la division : en Python 3, l'opérateur `//` n'existe pas, le seul opérateur de division est `/`. Si les deux opérandes (numérateur et dénominateur) sont de type `int`, alors la division est entière (comme `//` en Python 3) ; si au moins un des deux opérandes est de type `float`, alors l'opérateur `/` effectue une division réelle (comme `/` en Python 3).

**Prép. B.1.** En utilisant l'interpréteur `python2`, faites l'exercice 1.3.

**Prép. B.2.** En utilisant l'interpréteur `python2`, faites l'exercice 1.4.

**Prép. B.3.** En utilisant l'interpréteur `python2`, faites l'exercice 1.6.

**Prép. B.4.** Pour cette séance, vous allez avoir besoin de plusieurs images en format `png`. À l'aide de votre éditeur d'images préféré, créez des images en format `png`. Par exemple, GIMP est un excellent logiciel libre d'édition d'images. Entre autres, préparez des images de tailles  $500 \times 500$  pixels,  $800 \times 800$  pixels,  $800 \times 600$  pixels,  $600 \times 800$  pixels et  $100 \times 100$  pixels. N'oubliez pas d'apporter ces images au TP.

**Prép. B.5.** Que fait le code suivant ? Utilisez la fonction `help()` pour vous renseigner sur les fonctions utilisées.

```
import os

def test(x):
 return os.path.isfile(x) and x.split(".")[-1] == "png"

L = os.listdir(".")
L = list(filter(test, L))
print L
```

## B.1 Notions de base

Nous allons vous demander de vous familiariser avec une bibliothèque qui vous permet de créer des programmes de traitement d'images. Pour importer la bibliothèque, ouvrir une image et l'afficher, il suffit d'effectuer la suite d'instructions suivantes :

```
>>> import Image
>>> picture = Image.open(nom_de_fichier)
>>> picture.show()
```

**Ex. B.1.** Créez un dossier de travail pour la séance, mettez-y des images ; ouvrez et affichez celles-ci via l'interpréteur. Utilisez des images en .png. Vous remarquerez que le principe est le même que pour l'ouverture d'un fichier via la fonction `open()`.

**Ex. B.2.** Les images possèdent des informations comme leur taille, leur mode de couleur, leur format, etc. Pour les récupérer, il suffit de faire :

```
>>> picture.size
(2446, 3720)
```

Consultez de la même manière les attributs `mode`, `format`, `format_description`, `filename` et déterminez à quoi correspondent ces informations.

**Ex. B.3.** Les images sont enregistrées avec la méthode `save`, comme montré dans l'exemple suivant :

```
>>> picture.save(nom_de_fichier, format_du_fichier)
```

On peut créer ce qu'on appelle des *thumbnails* (c'est-à-dire des miniatures) sur base d'un tuple déterminant la taille de la miniature.

```
>>> size = (94, 94)
>>> picture.thumbnail(size)
```

Écrivez une fonction `traitement_par_lots(liste_noms, directory)` qui prend en paramètres une liste de noms de fichiers et un string contenant le chemin d'un dossier. La fonction devra ouvrir chaque fichier et en créer un *thumbnail* de taille  $128 \times 128$  qui sera sauvegardé dans le dossier dont le chemin est fourni par `directory`.

## B.2 Modifications et transformations géométriques

Vous pouvez également découper une image à partir d'un rectangle défini par les points supérieurs gauches et inférieurs droits. Ce rectangle est défini par un tuple contenant quatre valeurs (`x1`, `y1`, `x2`, `y2`) et doit être passé en paramètre à la méthode `crop(rectangle)`. Vous pouvez redimensionner l'image avec `resize(tuple)` et lui appliquer une rotation avec `rotate(degree)`. Vous pouvez appliquer des symétries orthogonales avec les appels à la méthode `transpose()` en lui passant, selon que vous désirez une symétrie d'axe vertical ou horizontal respectivement, `Image.FLIP_LEFT_RIGHT` ou `Image.FLIP_TOP_BOTTOM`. Voici un exemple :

```
>>> rectangle = (65, 65, 800, 800)
>>> picture = picture.crop(rectangle)
>>> picture = picture.rotate(90)
>>> picture = picture.transpose(Image.FLIP_TOP_BOTTOM)
```

**Ex. B.4.** Écrivez une fonction `traitement_miroir(liste_noms)` qui prend en paramètre une liste de noms de fichiers. Vous avez fait une séance de shooting photo, malheureusement l'angle de vue était faite sur un miroir, l'appareil incliné de  $30^\circ$ . Nous vous demandons de traiter toutes ces photos dont la liste des fichiers est passée en paramètre. Ces photos doivent être recadrées avec un rectangle  $(30, 30, 500, 500)$ , doivent être inversées par rapport à l'axe vertical et subir une rotation permettant de redresser l'image. Le résultat de ces manipulations pour chaque photo doivent remplacer le fichier d'origine.

### B.3 Filtres

Vous pouvez également appliquer différents filtres d'image à l'aide du module `ImageFilter`, dont voici un exemple :

```
import ImageFilter

picture2 = picture.filter(ImageFilter.BLUR)
```

Il existe beaucoup d'autres modules de retouche. Si vous voulez aller plus loin, vous pouvez vous documenter sur la bibliothèque disponible à l'adresse suivante :

<http://www.pythonware.com/library/pil/>

**Ex. B.5.** Jouez maintenant avec la documentation et les différents filtres. Étudiez les modules `ImageFont` et `ImageDraw` ; essayez de les utiliser pour ajouter du texte dans une image.