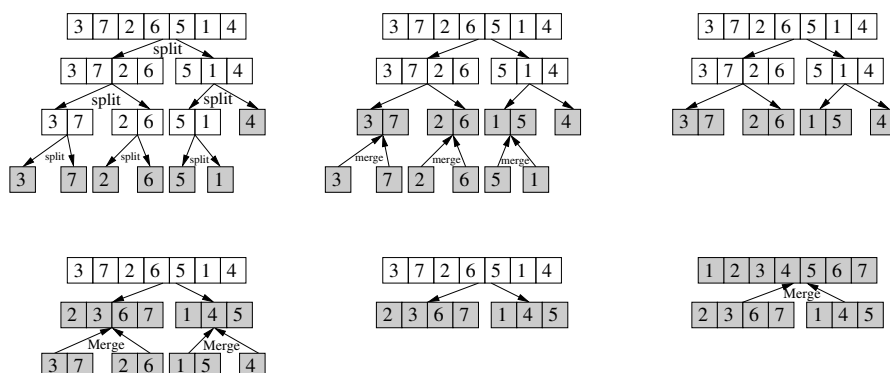


Programmation et Algorithmique I

Hadrien Mélot

Service d'Algorithmique, Département d'Informatique
Faculté des Sciences, Université de Mons



Remarques

Ce document est compilé automatiquement à partir des transparents du cours de Programmation et Algorithmique I : le contenu est donc identique aux transparents (disponibles sur *e-learning*) et seule la mise en page est différente. Néanmoins, les corrections ou modifications apportées aux transparents en cours d'année ne sont répercutées dans la version "syllabus" que lors de sa prochaine compilation (environ une fois par an).

Etant donné que le contenu de ce document est issu de transparents, tout n'y est donc pas toujours écrit dans le détail et il vous est conseillé de compléter ces notes avec vos notes personnelles ou vos lectures des livres de référence.

Si vous remarquez des erreurs ou des éléments mal retranscrits lors du passage automatique des transparents au syllabus, merci de les signaler !

Date de la compilation du présent document : 6 mai 2023.

Table des matières

1	Introduction	6
1.1	Objectifs du cours	6
1.2	Organisation pratique	6
1.3	Petite mise au point technique	7
1.4	Algorithmes et programmes	9
1.5	Erreurs (bugs)	14
1.6	Glossaire	15
2	Types, variables et expressions	17
2.1	Valeurs et types	17
2.2	Variables	18
2.3	Opérateurs et expressions	22
2.4	Un mot sur les fonctions	24
2.5	Erreurs fréquentes	25
2.6	Exercices interactifs	26
2.7	Glossaire	26
3	Fonctions	28
3.1	Appels de fonctions	28
3.2	Définir de nouvelles fonctions	31
3.3	Effets de bord	35
3.4	Flot d'exécution	38
3.5	Documenter son code	40
3.6	Un petit mot sur les Tuples (parenthèse)	42
3.7	Erreurs fréquentes	43
3.8	Exercices interactifs	43
3.9	Glossaire	43
4	Instructions conditionnelles	45
4.1	Expressions booléennes	45
4.2	Instructions conditionnelles	48
4.3	Fonctions booléennes	51
4.4	Entrées au clavier (parenthèse)	52
4.5	Conversion de réels en entiers (parenthèse)	54
4.6	Erreurs d'approximation (parenthèse)	55

4.7	Erreur fréquente	57
4.8	Exercices interactifs	58
4.9	Glossaire	58
5	Récurtivité	59
5.1	Récurtivité	59
5.2	Un langage complet (parenthèse)	65
5.3	Quelques algorithmes récursifs	66
5.4	Paramètres par défaut et arguments mots-clefs (parenthèse)	68
5.5	Erreurs fréquentes et debug	69
5.6	Exercice interactif	70
5.7	Glossaire	71
6	Itérations et Chaînes de caractères	72
6.1	Incrémenter et décrémenter	72
6.2	Boucles <code>while</code>	73
6.3	Un <code>str</code> est une séquence immuable de caractères	75
6.4	Boucles <code>for</code>	81
6.5	Invocation de méthodes sur les chaînes	82
6.6	Opérateurs sur les chaînes	85
6.7	Erreurs fréquentes et debug	86
6.8	Exercices interactifs : jouer avec les mots	88
6.9	Pour aller plus loin (à lire par soi-même)	90
6.10	Glossaire	92
7	Listes	93
7.1	Une liste est une séquence d'éléments	93
7.2	Listes et opérateurs	97
7.3	Méthodes sur les listes	98
7.4	Opérations fréquentes sur les listes	100
7.5	Objets et valeurs	103
7.6	Arguments de la ligne de commande (parenthèse)	107
7.7	Erreurs fréquentes et debug	108
7.8	Exercices interactifs	109
7.9	Pour aller plus loin (à lire par soi-même)	109
7.10	Glossaire	112
8	Prouver les propriétés des algorithmes	113
8.1	Une élection étrange	113
8.2	Preuve d'exactitude d'un algorithme itératif	116
8.3	Exemple 1 (boucle <code>while</code>) : la division euclidienne	117
8.4	Exemple 2 (boucle <code>while</code>) : calcul de l'exposant	119
8.5	Exemple 3 (liste et invariant graphique) : maximum d'une liste	121
8.6	Exemple 4 (boucle <code>for</code>) : somme d'une séquence	123
8.7	Exemple 5 (boucle <code>for</code>) : ajout des carrés	124
8.8	Exemple 6 (plusieurs boucles imbriquées) : tri par sélection	125
9	Complexité : évaluer l'efficacité des algorithmes	130

9.1	Efficacité des algorithmes	130
9.2	Calculer le temps CPU des algorithmes	131
9.3	Quelques nouveaux algorithmes	134
9.4	La notation grand- O et la notion de complexité dans le pire des cas	139
9.5	Evaluer la complexité des algorithmes itératifs	146
9.6	Evaluer la complexité des algorithmes récursifs	152
9.7	Analyse du comportement de nos algorithmes	155
9.8	Glossaire	158
10	Dictionnaires	159
10.1	Dictionnaires	159
10.2	Exemple d'utilisation des dictionnaires	162
10.3	Complexité des opérations sur les dictionnaires	162
10.4	Quelques algorithmes sur des dictionnaires	164
10.5	Glossaire	165
11	Tuples	166
11.1	Un tuple est une séquence immuable	166
11.2	Fonctions avec un nombre variable d'arguments	168
11.3	Tuples comme clefs d'un dictionnaire	170
11.4	Comparaison de tuples	170
11.5	Quelle séquence choisir?	170
11.6	Glossaire	171
12	Fichiers et exceptions	172
12.1	Lecture et écriture de fichiers	172
12.2	Sauvegarder des objets	175
12.3	Exceptions	176
12.4	Gérer les exceptions	177
12.5	Créer son propre type d'exception	180
12.6	A lire par soi-même (pour aller plus loin)	180
12.7	Glossaire	182
13	Introduction aux objets	184
13.1	La notion d'objet	184
13.2	Les attributs d'un objets et les méthodes <code>__init__</code> et <code>__str__</code>	186
13.3	Définir le comportement d'un objet via ses méthodes	191
13.4	Surcharge d'opérateurs	193
13.5	Remarques et précautions quand on manipule des objets	194
13.6	Un tout petit mot sur l'héritage	196
13.7	Illustrations de la POO	198
13.8	Glossaire	198

Introduction

Objectifs du cours • Organisation pratique • Petite mise au point technique • Algorithmes et programmes • Erreurs (bugs) • Glossaire

1.1. Objectifs du cours

Etre capable de

- comprendre un *algorithme*
- concevoir un *algorithme* (efficace)
- écrire un *programme*

1.2. Organisation pratique

- 30 heures de cours
- Transparents : <https://moodle.umons.ac.be>
- 60 heures de travaux pratiques : salles machines
- Examen écrit en janvier (Q1), juin (Q2, uniquement pour B1) et septembre (Q3)
- Test écrit en milieu de Q1

Livres de références

- DOWNEY, A., *Think Python : how to think like a computer scientist*, Green Tea Press (2nd edition, 2015)
www.greenteapress.com/thinkpython2/thinkpython2.pdf
- AHO, A. et ULLMAN, J., *Concepts fondamentaux de l'Informatique*, Dunod (1993)

Equipe enseignante

Titulaire : Hadrien MÉLOT

Assistants : Sébastien BONTE, Jeremy DUBRULLE et Pierre HAUWEELE

Bureaux : 2ième étage du De Vinci

Emails : prenom.nom@umons.ac.be

Mise en garde concernant les TPs

- L'algorithmique et un langage de programmation, cela s'apprend de manière *active* et avec un travail *personnel* et *régulier*.
- *Avant* d'aller à une séance de TP, bien relire le cours.
- *Erreur fatale* : croire que les TPs vont permettre de comprendre le cours. Les TPs supposent que vous ayez *compris*, pour permettre d'*aller plus loin*.
- Si vous vous sentez perdus, contactez-nous ou pensez au *tutorat* (voir horaire).

⇒ voir document "organisation et consignes"

1.3. Petite mise au point technique

Quelques notions techniques importantes pour un programmeur

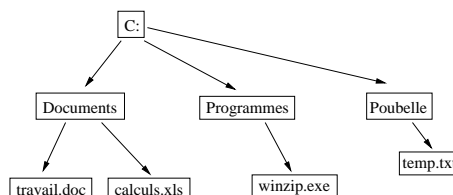
1. Hiérarchie des fichiers (vision graphique et console)
2. Commandes de base dans la console
3. Distinction entre fichiers "texte" et "binaire"

⇒ Ces notions vont être expliquées brièvement ci-après.

Vous vous familiariserez rapidement avec celles-ci lors des Travaux Pratiques.

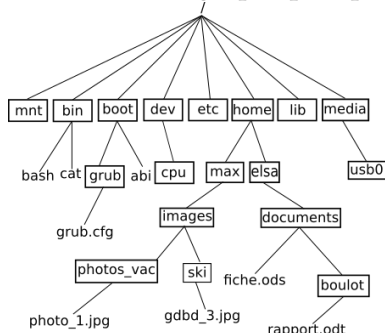
Voir également le tutoriel "ligne de commande" disponible sur moodle.

Hiérarchie des fichiers



Illustré en séance :

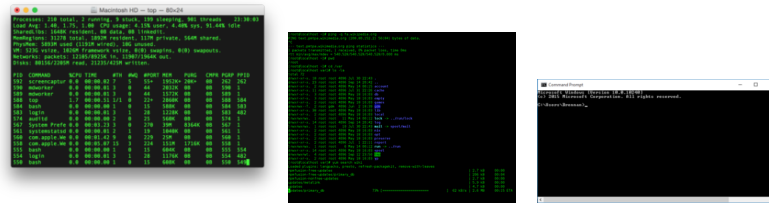
- notion de racine, de répertoire "home" et de répertoire "courant" ;
- vision "graphique" pour voyager dans cette hiérarchie.



Source image de droite : https://pixees.fr/informatiquelycee/n_site/nsi_prem_cmd_base_linux.html

Commandes de base dans la console

Tous les systèmes d'exploitation (OS X, Linux, Unix, Windows, etc.) possèdent une (ou plusieurs) *console(s)* qui permettent de lancer des commandes sans passer par l'interface graphique.



Noms des consoles : terminal (Mac), invite de commande ou command prompt (Windows), shell (Unix), le nom d'un certain type de console (bash, zsh, etc.), ...

Source des images : <https://fr.wikipedia.org>

Pour connaître le répertoire courant (souvent le « home » par défaut quand on lance la console), on utilise la commande

`pwd`

sous Linux et Mac ; et la commande

`cd`

sous Windows.

Remarque : la commande `cd` sans argument a un autre comportement sous Linux et Mac, voir ci-après.

Exemple :

```
hmelot@Opeth ~ % pwd
/Users/hmelot
```

Pour lister les fichiers et les répertoires contenus dans le répertoire courant, on utilise la commande

`ls`

sous Linux et Mac ; et la commande

`dir`

sous Windows.

Pour changer de répertoire, on utilise la commande

`cd NomRepertoire`

ou

`cd Chemin/Vers/Repertoire`

Cas spéciaux :

- « `cd ..` » pour « remonter » d'un répertoire
- « `cd` » pour se placer dans le « Home » ; « `cd /` » pour la racine (Linux et Mac uniquement)

⇒ *illustration de la vision "console" pour voyager dans la hiérarchie des fichiers*

Distinction entre fichiers “texte” et “binaire”

Un fichier de *texte* (ou fichier texte brut) ne contient que des caractères (il utilise pour cela une forme de codage comme le code ASCII).

Un fichier *binaire* est un fichier qui n’est pas interprétable directement sous forme de texte. Il utilise un codage binaire pour stocker ses données (il peut cependant y avoir aussi des caractères dans ce type de fichiers). Par exemple, un fichier `.mp3` est un fichier binaire stockant des informations pour représenter un morceau de musique, un fichier `.jpg` pour représenter une image. Il faut utiliser un programme spécifique, qui connaît le format, pour décoder ce type de fichiers.

Il existe des “éditeurs de texte” (par ex., `gedit`, `notepad`) pour les fichiers “texte”.

Question : d’après vous, un fichier `Word` ou `OpenOffice` est un fichier texte ou binaire ?

Ce sont des fichiers binaires ! Pour les coder et les décoder, on doit utiliser un programme spécifique (programmes de *traitement* de texte). Ils ne contiennent pas que du texte, ils codent également toute une série d’informations relatives au *format* et à la manière de *présenter* le document, notamment.

⇒ *illustration via la commande `more` (fonctionne sous Linux, Mac et Windows)*

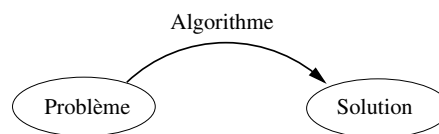
On écrit du code informatique dans des fichiers texte. Certains éditeurs de texte reconnaissent (via l’extension du fichier) qu’il s’agit de code et ajoutent une *coloration syntaxique* pratique pour le programmeur. Mais en réalité, le fichier de code est bien un fichier de texte brut.

Illustration : création d’un fichier `hello.py` qui contient une ligne de code en Python et comparaison des vues via la commande `more` et via un éditeur de code.

1.4. Algorithmes et programmes

Qu’est-ce qu’un algorithme ?

Tâche principale de l’informaticien : *résoudre des problèmes*



1. spécification du problème
 - qu’a-t-on besoin pour résoudre le problème (*entrées*) ?
 - que doit-on fournir comme solution (*sorties*) ?
2. résolution : comment trouver la solution (méthode, *algorithme*) ?

Définition. Un *algorithme* est une séquence d’étapes précises et non ambiguës pouvant être exécutées de façon automatique.

Exemples d’algorithmes

Spécification du problème :

Entrée : L'heure h (entier allant de 0 à 23)

Sortie : Un message de bienvenue spécifique

Algorithme :

- 1: **si** $h \leq 6$ **alors**
- 2: dire "Bonne nuit"
- 3: **sinon si** $h \leq 18$ **alors**
- 4: dire "Bonjour"
- 5: **sinon**
- 6: dire "Bonsoir"
- 7: **fin si**

Spécification du problème :

Entrée : Un dictionnaire et un mot

Sortie : La définition du mot recherché

Algorithme :

- 1: lire le premier mot du dictionnaire
- 2: **tant que** le mot lu n'est pas le mot recherché **faire**
- 3: lire le mot suivant
- 4: **fin tant que**
- 5: lire la définition du mot recherché

De manière plus efficace :

- 1: ouvrir le dictionnaire au milieu
- 2: **tant que** la page courante ne contient pas le mot **faire**
- 3: ouvrir le dictionnaire au milieu de la partie restante
- 4: **fin tant que**
- 5: **rechercher** la définition du mot dans la page courante

De manière plus précise et moins ambiguë :

- 1: $pageDebut \leftarrow 1$
- 2: $pageFin \leftarrow$ numéro de la dernière page du dictionnaire
- 3: $pageCourante \leftarrow pageDebut + (pageFin - pageDebut)/2$
- 4: **tant que** $pageCourante$ ne contient pas le mot **faire**
- 5: **si** le mot se trouve **avant** $pageCourante$ **alors**
- 6: $pageFin \leftarrow pageCourante - 1$
- 7: **sinon**
- 8: $pageDebut \leftarrow pageCourante + 1$
- 9: **fin si**
- 10: $pageCourante \leftarrow pageDebut + (pageFin - pageDebut)/2$
- 11: **fin tant que**
- 12: **rechercher** la définition du mot dans $pageCourante$

Qu'est-ce qu'un programme ?

Définition. Un *programme* est une séquence d'instructions qui spécifie comment réaliser un calcul ou une tâche. Ils sont décrits à l'aide de *langages de programmation*.

Remarque : le mot "programme" est utilisé aussi pour l'application qui "tourne" sur la machine

Beaucoup de langages différents mais les instructions permettent par exemple de :

- réaliser des opérations mathématiques (par ex. addition, multiplication)
- obtenir des données au clavier, depuis un fichier, etc.
- afficher des données à l'écran ou écrire des données dans un fichier
- exécution conditionnelle : vérifier si certaines conditions sont respectées et exécuter la séquence d'instructions appropriée
- réaliser une tâche de manière répétitive

Langages de programmation

On distingue les langages

- de *haut niveau* : compréhensible par l'humain (souvent langage formel avec des mots issus de l'anglais) : Python, Java, C, C++, PHP, Scheme, Pascal, etc.
- de *bas niveau* : exécutables par un ordinateur (langages machine ou assembleur)

Avantages d'un programme écrit dans un langage de haut niveau :

- beaucoup plus facile à écrire et à (re)lire ;
- plus rapide à écrire ;
- plus court ;
- *portable*
 - peut être utilisé sur différents types d'ordinateurs avec peu ou pas de modifications ;
 - un programme de bas niveau n'est exécutable que sur un type bien particulier d'ordinateurs et doit être réécrit pour d'autres.

Remarque. La grande majorité des programmes sont écrits en langages de haut niveau (sauf quelques applications spécialisées)

En pratique. Un programme compréhensible par l'humain (haut niveau) est traduit en un programme compréhensible par la machine (bas niveau) de manière automatique (par un programme dédié à cette tâche)

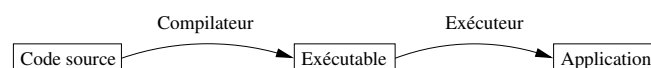
2 types de programmes "traducteurs" :

- compilateurs
- interpréteurs

Compilateurs

Définition. Un *compilateur* est un programme qui lit un *code source* et le traduit en un *exécutable*, avant l'exécution du programme.

- *code source* : programme écrit dans un langage de haut niveau spécifique
- *exécutable* : code compilé qui peut être exécuté sans être traduit à nouveau
- *application* : exécution du programme (qui "tourne" sur la machine)



Exemple. Programme écrit en C, compilé avec `gcc` et exécuté dans une console.

```

#include <stdio.h>
#include <time.h>

int main() {
    time_t timestamp = time(NULL);
    struct tm * t = localtime(&timestamp);
    int hour = t->tm_hour;

    if (hour <= 6)
        printf("Bonne nuit. ");
    else if (hour <= 18)
        printf("Bonjour. ");
    else
        printf("Bonsoir");

    printf("Il est %02uh %02umin %02usec.\n", t->tm_hour, t->tm_min, t->tm_sec);

    return 0;
}

```

Exemple. Programme écrit en Java, compilé avec `javac` et exécuté dans une console (avec `java`).

```

import java.util.Calendar;

public class Greeter {
    private Calendar d;

    public Greeter() {
        updateTime();
    }

    public void updateTime() {
        d = Calendar.getInstance();
    }

    public void sayHello() {
        updateTime();
        int hour = d.get(Calendar.HOUR_OF_DAY);

        if (hour <= 6)
            System.out.print("Bonne nuit. ");
        else if (hour <= 18)
            System.out.print("Bonjour. ");
        else
            System.out.print("Bonsoir. ");

        System.out.println("Il est " + hour + "h " + d.get(Calendar.MINUTE)
            + "min " + d.get(Calendar.SECOND) + "sec");
    }

    public static void main(String[] args) {
        Greeter g = new Greeter();
        g.sayHello();
    }
}

```

Interpréteurs

Définition. Un *interpréteur* est un programme qui lit un *code source* et l'exécute pas à pas.



Exemple. Programme interprété par Python des 2 façons suivantes :

- *mode script* (à partir d'un fichier)

- *mode interactif* (dans une console ou dans *IDLE*)

Python : mode script

Fichier `say_hello.py`

```
from datetime import datetime

d = datetime.now()
hour = d.hour

if hour <= 6:
    print('Bonne nuit. ', end='')
elif hour <= 18:
    print('Bonjour. ', end='')
else:
    print('Bonsoir. ', end='')

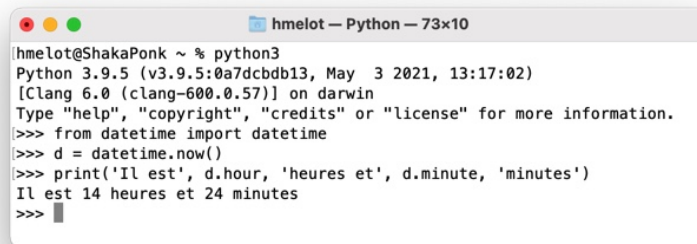
print('Il est', d.hour, 'h', d.minute, 'min', d.second, 'sec')
```

A lancer avec la commande `python3 say_hello.py`

Bonjour. Il est 10 h 42 min 13 sec

Python : mode interactif

- Le mode interactif se lance en tapant simplement `python3` dans la console
- Les commandes (prompt) ou *instructions*



```
hmelot@ShakaPonk ~ % python3
Python 3.9.5 (v3.9.5:0a7dcdb13, May 3 2021, 13:17:02)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from datetime import datetime
>>> d = datetime.now()
>>> print('Il est', d.hour, 'heures et', d.minute, 'minutes')
Il est 14 heures et 24 minutes
>>>
```

Le mode interactif est pratique pour tester des petites choses, pas pour écrire un programme élaboré.

Choix du langage pour le cours

- Le langage utilisé dans ce cours pour *implémenter* les algorithmes sera le *Python* (version 3)
- *Attention* : la version 2 est incompatible!
- Sur moodle : différences principales entre Python 2 et Python 3 (si vous consultez des examens antérieurs à septembre 2016) *Exemple* : en Python 2, il n'est pas nécessaire d'utiliser des parenthèses avec `print`.
- Si possible : *installez Python 3 chez vous* au plus vite
- <https://www.python.org> (téléchargement, documentation, etc.)

Pourquoi Python ?

Principalement car il est un langage *simple* au niveau de la syntaxe, et donc bien adapté d'un point de vue *pédagogique*.

Ce n'est pas seulement un langage pédagogique. Il est utilisé par beaucoup d'entreprises dont Yahoo, Google, Walt Disney, la NASA, etc.

<https://wiki.python.org/moin/OrganizationsUsingPython>

1.5. Erreurs (bugs)

Les erreurs sont fréquentes quand on programme.

Trois sortes d'erreurs :

- erreurs de *syntaxe*
- erreurs à l'exécution (ou *exceptions*)
- erreurs *sémantiques* (ou logiques)

Deboguer (debug) : processus de repérage et correction des erreurs

Exemple de programme sans erreur :

```
>>> print('La somme des deux premiers nombres entiers positifs est', (1+2))  
  
La somme des deux premiers nombres entiers positifs est 3
```

Erreurs de syntaxe

Définition. Une *erreur de syntaxe* est une violation des règles du langage de programmation. Celle-ci est signalée par le compilateur ou l'interpréteur lors de son analyse (*parsing*).

Exemple (manque une parenthèse) :

```
>>> print('La somme des deux premiers nombres entiers positifs est', 1+2))  
  
SyntaxError: invalid syntax
```

Exemple (manque une apostrophe) :

```
>>> print('La somme des deux premiers nombres entiers positifs est, (1+2))  
  
SyntaxError: EOL while scanning string literal
```

Exceptions

Définition. Une *exception* (ou erreur à l'exécution) est une erreur qui apparaît après le lancement du programme quand quelque chose d'exceptionnel se produit et qui n'est pas du à une violation de la syntaxe.

Exemples : ouverture d'un fichier sur lequel l'utilisateur n'a pas les droits d'accès, perte de la connexion réseau, etc.

Erreurs sémantiques

Définition. Une *erreur sémantique* (ou erreur logique) se produit quand le programme ne donne pas le résultat attendu.

- apparaît quand le programme est syntaxiquement correct mais la logique des instructions est erronée
- peut être difficile à identifier et corriger (travail de "détective" : savoir déboguer est une importante qualité du programmeur)

Exemples :

```
>>> print('La somme des deux premiers nombres entiers positifs est', (1+3))
```

```
La somme des deux premiers nombres entiers positifs est 4
```

```
>>> print('La somme des trois premiers nombres entiers positifs est', (1+2))
```

```
La somme des trois premiers nombres entiers positifs est 3
```

1.6. Glossaire

résolution de problème : processus de formulation d'un problème (spécification), trouver une solution et exprimer la solution

algorithme : séquence d'étapes précises et non ambiguës pouvant être exécutées de façon automatique

programme : séquence d'instructions écrites dans un langage de programmation particulier et qui spécifie comment réaliser un calcul ou une tâche

langage de haut niveau : un langage comme Python facile à lire et écrire pour l'humain

langage de bas niveau : un langage exécutable par un ordinateur (aussi appelé langage machine ou assembleur)

portabilité : le fait qu'un programme puisse être exécuté sur plus d'une sorte d'ordinateurs

interpréter : exécuter un programme écrit dans un langage de haut niveau en le traduisant pas à pas

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

code source : un programme écrit en langage de haut niveau avant sa compilation ou son interprétation

exécutable : un programme après compilation, prêt à être exécuté

prompt : les caractères >>> qui indiquent que l'interpréteur est prêt à recevoir des instructions

script : un programme stocké dans un fichier (en vue d'être interprété)

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

bug : une erreur dans un programme

deboguer : le processus d'identification et de correction des 3 types d'erreurs de programmation

syntaxe : la structure et les règles d'un langage de programmation

erreur de syntaxe : une violation des règles d'écriture dans un programme qui le rend impossible à interpréter ou à compiler

exception : une erreur détectée pendant l'exécution du programme

sémantique : la signification (la logique, le sens) d'un programme

erreur sémantique : une erreur dans un programme qui fait que quelque chose de différent de ce que le programmeur voulait réaliser se produit

parser : examiner un programme et analyser sa structure syntaxique

print : fonction (voir ch. 3) utilisée pour afficher une valeur à l'écran

Types, variables et expressions

Valeurs et types • Variables • Opérateurs et expressions • Un mot sur les fonctions • Erreurs fréquentes • Exercices interactifs • Glossaire

Un peu de mentalisme

Avant d’aborder ce chapitre, je vous propose une petite séance de *mentalisme*.



Source de l'image : <https://www.artesine.fr>

2.1. Valeurs et types

Une *valeur* est un des éléments de base d’un programme, comme un nombre ou un mot.
Exemples : 1, 2.5 ou 'Bonjour'

Toute valeur possède un *type* particulier.

- 1 et 2 sont des entiers (*integer*) : `int`
- 1.4 et 2.0 sont des réels (nombres à virgule flottante, *floating-point*) : `float`
- 'Bonjour' est une chaîne de caractères (*string*), identifiable (par vous et l’interpréteur Python) grâce aux apostrophes : `str`

(Un type de valeurs \simeq *classe* de valeurs d'un même type. Il y a beaucoup de types différents et vous apprendrez à créer les vôtres!)

L'interpréteur peut vous donner le *type* d'une valeur :

```
>>> type(4)
<class 'int'>
>>> type('Bonjour')
<class 'str'>
>>> type(1.2)
<class 'float'>
```

Remarque : `type` est une *fonction* prédéfinie de Python. Un appel à une fonction est visible par l'utilisation des parenthèses. On reviendra sur la notion de fonction plus loin dans ce chapitre et le suivant.

Il y a des différences syntaxiques pour que Python reconnaisse les types :

- le point détermine un `float`
- les apostrophes (ou les guillemets) déterminent un `str`

```
>>> type(2)
<class 'int'>
>>> type(2.)
<class 'float'>
>>> type('2')
<class 'str'>
>>> type("2")
<class 'str'>
```

2.2. Variables

Une *variable* est un nom (symbolique) qui permet de faire référence à une valeur (et de la stocker en mémoire).

Une *assignation* est une instruction qui permet de créer une nouvelle variable et de lui attribuer une valeur.

Syntaxe d'une assignation : `nom = valeur`

```
>>> i = 12
>>> message = 'La valeur de PI est'
>>> pi = 3.14159265
```

Une fois assignée, on peut retrouver et utiliser la valeur stockée en mémoire en utilisant le nom de la variable.

En Python, on peut *afficher* le contenu d'une (ou plusieurs) variable(s)

- à l'aide de la fonction `print` (mode script et interactif)
- en donnant simplement le nom d'une variable (uniquement en mode interactif)

```
>>> print(i)
12
```

```
>>> print(message, pi)
La valeur de PI est 3.14159265
>>> i
12
```

Remarque : en mode script, la fonction `print` est nécessaire pour afficher une variable (testez-le!).

Tant qu’une variable n’est pas assignée, elle n’existe pas et on ne peut donc pas afficher sa valeur.

```
>>> print(j)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined
>>> j = 18
>>> print(j)
18
```

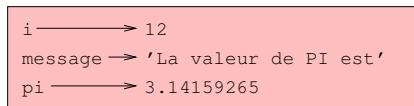
Remarque : le message d’erreur ci-dessus est pour le moment un peu cryptique mais la dernière ligne de celui-ci est explicite : `NameError: name 'j' is not defined`

Une *assignation* permet également de mettre à jour la valeur d’une variable déjà créée.

```
>>> print(i)
12
>>> i = 5
>>> print(i)
5
```

Visualisation de l’état des variables

On peut représenter les variables par un *diagramme d’état*.



Vous pouvez également utiliser *Python Tutor* pour suivre pas à pas la “vie” de vos variables en mémoire.

<https://pythontutor.com/visualize.html>

Ces méthodes visuelles seront bien utiles pour comprendre certaines subtilités!

Visualisation de l’état des variables

Démonstration : Python Tutor

Python 3.6
([known limitations](#))

```

1 i = 42
2 x = 2.6175
→ 3 print(i)
→ 4 i = 9
5 msg = 'hello'
6 print(i, x, msg)

```

[Edit this code](#)

→ line that just executed
→ next line to execute

Step 4 of 6

Print output (drag lower right corner to resize)

42

Frames Objects

Global frame	
i	42
x	2.6175

Type d'une variable

Le type d'une variable est le type de la valeur qu'elle réfère.

```

>>> type(i)
<class 'int'>

>>> type(message)
<class 'str'>

>>> type(pi)
<class 'float'>

```

Précisions à propos de la syntaxe d'une assignation

Rappel de la syntaxe d'une assignation :

nom = valeur

A gauche (nom) Vous avez (presque) toute liberté pour nommer votre variable *A droite* (valeur) Vous pouvez coder tout ce que vous voulez tant que Python peut en déduire une valeur

Le nom d'une variable

- Contraintes :
 - ne peut contenir que des lettres (minuscules et majuscules), des chiffres et le caractère `_` (underscore)
 - *doit* commencer par une lettre ou le caractère `_`
- Conventions :
 - devrait avoir du sens (dire ce qu'elle représente)
 - devrait commencer par une lettre minuscule

Attention à la *casse* : `pi` est différent de `Pi` ou de `PI`

Exemples de noms valides : `x1`, `maVariable`, `mon_entier`

Exemples de noms invalides : `2be`, `x@`, `class`

Pourquoi `class` n'est-il pas valide ?

On ne peut pas utiliser `class` pour une variable car c'est un mot-clef (mot réservé) du

langage Python.

Python 3 possède 33 mots-clef :

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Si l'interpréteur se plaint du nom d'une de vos variables, consultez cette liste !

Une valeur (à droite dans l'assignation) peut être obtenue

- en la donnant directement (ce qu'on a fait jusqu'ici, mais si on se limitait à ça, nos programmes ne seraient pas très impressionnants!)
- via le nom d'une variable
 - cela peut être une autre variable déjà définie
 - ou la variable elle-même si elle a déjà été définie auparavant
- via le résultat d'une expression arithmétique
- et via d'autres mécanismes que vous découvrirez petit à petit comme le résultat d'une fonction qui peut cacher un algorithme aussi complexe que possible

Il est possible de combiner tout cela tant qu'au final Python peut en déduire une valeur !

Exemples

```
>>> a = 13
>>> b = a
>>> c = 3 + 2
>>> d = b + c
>>> a = a + 1
>>> print(a, b, c, d)

14 13 5 18
```

Remarque : notez que même si `b` a été initialisé avec la valeur de `a`, ces deux variables n'ont plus la même valeur à la fin de ces instructions. Comprenez-vous pourquoi ?

Votre premier algorithme

Problème. Vous avez deux variables `a` et `b` en entrées. Comment faire pour échanger leurs valeurs ?

```
>>> a = 17
>>> b = 21
>>> ???
>>> print(a)

21

>>> print(b)

17
```

(Utilisation de PythonTutor pour tester les solutions proposées par les étudiants)

Solution : utiliser une troisième variable temporaire pour ne pas perdre de valeur

```
>>> a = 17
>>> b = 21

>>> a_old = a
```

```
>>> a = b

>>> b = a_old
>>> print(a)
21

>>> print(b)
17
```

a	→	17
b	→	21

a	→	17
b	→	21
a_old	→	17

a	→	21
b	→	21
a_old	→	17

a	→	21
b	→	17
a_old	→	17

2.3. Opérateurs et expressions

Opérateurs et expressions

Python supporte toute une série d'opérateurs arithmétiques pour créer des expressions. On a déjà vu l'addition. Le symbole `*` est utilisé pour la multiplication.

```
>>> pi = 3.14159265
>>> rayon = 10
>>> circonference = 2 * pi * rayon
>>> aire = pi * rayon * rayon
>>> print(circonference)
62.831853

>>> print(aire)
314.59265
```

Opérateurs

Les *opérateurs* sont des symboles spéciaux qui représentent des opérateurs arithmétiques comme l'addition ou la division. Les *opérandes* sont des valeurs de type `int` ou `float`. Comme pour l'assignation, cela ne doit pas forcément être une valeur explicitement donnée, du moment que Python peut en déduire une valeur.

- `+` → addition (`20 + 32`)
- `-` → soustraction (`hour - 1`)
- `*` → multiplication (`hour * 60 + minute`)
- `/` → division (`minute / 60`)
- `**` → exposant (`5 ** 2`)

- `//` → division entière (`7 // 2`)
- `%` → modulo (`7 % 2`) : reste de la division entière

Les parenthèses peuvent également être utilisées :

```
(5 + 9) * (15 - 7)
```

Division de deux entiers et Division entière

Division de deux entiers : retourne une valeur de type `float`

```
>>> 7 / 2
3.5
>>> 4 / 2
2.0
```

Division entière : retourne une valeur de type `int` qui est la partie entière du résultat.

```
>>> 7 // 2
3
>>> 7 % 2
1
```

Remarque : comportement très différent en Python 2 (voir note sur moodle)

Expressions

Une *expression* est une combinaison de valeurs, variables et opérateurs.

Exemples :

- `17`
- `x`
- `x + 17`

En mode interactif, l'interpréteur *évalue* l'expression et affiche le résultat.

```
>>> 1 + 1
2
```

Par contre, dans un script, une expression est évaluée mais le résultat n'est pas affiché : on peut croire qu'il ne se passe rien si on ne demande pas d'afficher le résultat !

Règles de précedence

L'ordre de précedence des opérateurs est le même qu'en mathématiques :

1. parenthèses (donc `2 * (3 - 1)` donne 4)
2. exposant (donc `2 ** 1 + 1` donne 3 et `3 * 1 ** 3` donne 3 et pas 27)
3. multiplication et division (donc `2 * 3 - 1` donne 5)
4. addition et soustraction

Les opérateurs ayant la même précedence sont évalués de gauche à droite.

Exemple : `d / 2 * pi` : la division est effectuée en premier, et le résultat est donc $\frac{d \cdot \pi}{2}$.

Pour obtenir $\frac{d}{2\pi}$, il faut écrire `d / (2 * pi)`

Opérations sur les chaînes de caractères

On ne peut pas utiliser tous les opérateurs mathématiques sur des opérandes de type `str`.

Opérateurs acceptés pour les chaînes

- l'opérateur `+` permet de *concaténer* deux `str` (attacher le deuxième `str` au premier)

```
>>> first = 'chat'
>>> second = 'eau'
>>> together = first + second
>>> print(together)
chateau
```
- l'opérateur `*` utilisé sur un `str` et un `int` permet de *répéter* le `str`

```
>>> 'ha' * 3
'hahaha'
```

2.4. Un mot sur les fonctions

Réutiliser un calcul

Soit le code suivant :

```
>>> pi = 3.14159265
>>> rayon = 10
>>> circonference = 2 * pi * rayon
>>> print(circonference)
62.831853

>>> rayon = 2
>>> print(rayon)
2
```

D'après vous, quelle valeur sera affichée si on affiche à nouveau `circonference`? La circonférence d'un cercle de rayon 10 ($\simeq 62.83$) ou celle d'un cercle de rayon 2 ($\simeq 12.57$)?

Bien que le rayon ait été modifié, la circonférence n'est pas mise à jour automatiquement. Il faut recalculer la nouvelle circonférence.

```
>>> print(circonference)
62.831853

>>> circonference = 2 * pi * rayon
>>> print(circonference)
12.5663706
```

Ce comportement est normal au vu du mécanisme de l'assignation :

\implies *démonstration via Python Tutor*

Devoir « copier / coller » le calcul de la circonférence n'est pas très élégant! Heureusement, nous pouvons plutôt définir `circonference` comme une *fonction* qui prend en paramètre le rayon et cache le code du calcul de la circonférence.

```
pi = 3.14159265

def circonference(rayon):
    resultat = 2 * pi * rayon
    return resultat
```



```
circ1 = circonference(10)
circ2 = circonference(2)

print('Circonference: cercle1 =', circ1, ' cercle2 =', circ2)
```

⇒ *exécution de ce script et démonstration via Python Tutor*

Définir une fonction

```
def circonference(rayon):
    resultat = 2 * pi * rayon
    return resultat
```

Remarques :

- `def` est un mot-clef qui introduit la définition d'une fonction
- il est suivi du nom de la fonction (même contraintes que pour les noms de variables); puis d'une liste d'arguments (séparés par des virgules) entre parenthèses (ici, il n'y en a qu'un); et enfin du symbole « : »
- le code de la fonction est *indenté* (attention : toujours le même nombre d'espaces)
- on retourne la valeur souhaitée via `return`

Les fonctions

On reconnaît l'appel à une fonction via les parenthèses et, quand elles retournent une valeur, elles peuvent être utilisées dans le code où une telle valeur est attendue, comme dans une assignation, une expression ou un argument d'une autre fonction.

Exemples :

```
x = circonference(10)
y = circonference(10) + circonference(5)
z = circonference(circonference(10))
```

Il y a toute une série de fonctions déjà prédéfinies en Python, comme `type` et `print`. D'autres sont disponibles dans des « modules » qu'il faut importer.

Ces modules contiennent parfois aussi des variables. C'est le cas pour π que nous ne devons donc plus redéfinir nous même dorénavant.

```
>>> import math
>>> math.cos(0)
1.0
>>> math.pi
3.141592653589793
>>> math.cos(math.pi)
-1.0
```

Le chapitre suivant revient en détails sur les fonctions.

2.5. Erreurs fréquentes

- utiliser un mot-clef comme nom de variable (par ex. `class`)
- mettre un espace dans le nom d'une variable (par ex. `bad name`)
- utiliser une variable avant de l'avoir créée (rappel : pour créer une variable, la placer à gauche d'une assignation)

```
>>> principal = 257.50
>>> interet = principal * taux
Traceback (most recent call last):
  File "<pyshell\#88>", line 1, in <module>
    interet = principal * taux
NameError: name 'taux' is not defined
```

- ne pas respecter la “casse” de caractères : par ex., `pi` est différent de `Pi`
- erreur sémantique sur l’ordre des opérations : par ex., écrire `1.0 / 2.0 * pi` pour $\frac{1}{2\pi}$

2.6. Exercices interactifs

Exercice. On a vu que `n = 42` est une instruction valide. Qu’en est-il de `42 = n`?

Exercice. Que se passe-t-il avec `x = y = 1`?

Exercice. En math, on peut multiplier `x` et `y` de la façon suivante : `xy`. Que se passe-t-il si on essaye ça en Python?

Exercice. Supposons que nous exécutons les instructions suivantes.

```
largeur = 17
hauteur = 12
letter = 'x'
```

Pour chacune des expressions suivantes, quel est la *valeur* de l’expression et son *type*?

- `largeur / 2`
- `largeur // 2`
- `hauteur / 3`
- `1 + 2 * 5`
- `letter * 5`

Exercice. Il est 19h47 et vous vous apprêtez à regarder la version longue de *La Communauté de l’Anneau* qui dure 3 heures et 48 minutes. A quelle heure votre film sera-t-il terminé?

Ecrivez un script qui répond à cette question.

2.7. Glossaire

valeur : unité élémentaire de données, comme un nombre ou une chaîne de caractères, qu’un programme manipule.

type : catégorie de valeurs. Les types vus jusqu’à présent : entiers (`int`), réels (`float`), et chaînes de caractères (`str`).

variable : nom qui réfère à une valeur.

diagramme d’état : représentation graphique d’un ensemble de variables et des valeurs qu’elles réfèrent.

assignation : instruction qui assigne une valeur à une variable (`variable = valeur`)

opérateur : symbole spécial qui représente une opération simple comme l’addition, la multiplication ou la concaténation de chaînes de caractères.

opérande : une des valeurs sur lesquelles un opérateur s’applique.

division entière : opération qui divise deux nombres entiers et retourne un entier

(uniquement la partie entière du résultat).

expression : combinaison de variables, d'opérateurs et de valeurs et dont le résultat est une valeur.

évaluer : simplifier une expression en appliquant les opérations dans le but d'obtenir la valeur résultat.

règles de précedence : ensemble de règles définissant l'ordre dans lequel les expressions impliquant plusieurs opérateurs et opérandes sont évaluées.

concaténer : joindre bout à bout des chaînes de caractères.

mot-clef : mot réservé et utilisé par le langage Python pour parser un programme ; on ne peut pas utiliser les mots-clef comme nom de variables.

Fonctions

Appels de fonctions • Définir de nouvelles fonctions • Effets de bord • Flot d'exécution • Documenter son code • Un petit mot sur les Tuples (parenthèse) • Erreurs fréquentes • Exercices interactifs • Glossaire

3.1. Appels de fonctions

Faire appel à une fonction

Définition. Une *fonction* est une séquence d'instructions à laquelle on donne un nom. Elle peut prendre en entrée une liste d'*arguments* et fournit en sortie une *valeur de retour*.

On peut voir une fonction comme une *boîte noire* qui effectue un travail.

- les arguments sont ce qu'elle a besoin pour faire son travail (entrées)
- la valeur de retour est le résultat de son travail

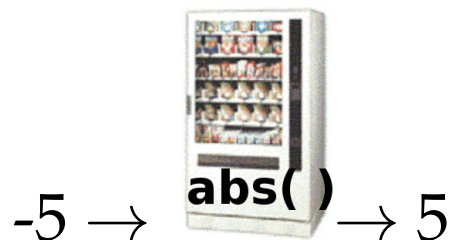


- on reconnaît l'appel à une fonction par l'utilisation de parenthèses qui entourent les arguments
- on dit qu'une fonction "prend" quelque chose comme argument et "retourne" un résultat

Exemple : la fonction `abs` prend un nombre en entrée et retourne sa valeur absolue.

```
>>> x = -5
>>> y = abs(x)
>>> print(y)
```

```
5
>>> print(abs(8.0))
8.0
```



Exemple : la fonction `print` affiche les valeurs de ses arguments (elle n'a pas pour but de retourner quelque chose)

Exemple : la fonction `int` convertit une chaîne en entier

```
>>> int('32')
32
>>> s = '100' + '20'
>>> x = int(s)
>>> print(x)
10020
```



Remarque : `print` est une fonction en Python 3 (parenthèses nécessaires) mais en Python 2, c'est une instruction (pas de parenthèses nécessaires).

Fonctions de conversion de types

Python fournit une série de fonctions permettant de convertir le type d'une valeur.

- `int()` : chaîne de caractères → entier : ok si la chaîne représente un entier

```
>>> int('32')
32
>>> int('Bonjour')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Bonjour'
```

- `int()` : réel → entier : ne garde que la partie entière

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

- `float()` : entier ou chaîne → réel : ok si la chaîne représente un réel

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

- `str()` : entier ou réel → chaîne

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

Modules

Définition. Un *module* est un fichier qui contient une collection de fonctions et variables reliées (ainsi que d'autres choses, voir plus tard). Il faut l'importer pour l'utiliser.

Pour accéder aux fonctions (*appels* de fonctions) ou aux variables définies dans un module, on utilise la notation "*point*" :

```
>>> import math
>>> x = math.sqrt(9)
>>> print(x)
```

3.0

```
>>> print(math.pi)
```

3.141592653589793

Pour éviter de devoir donner le nom du module lors d'un appel à une fonction, on peut importer nommément une fonction.

```
>>> from math import sqrt
>>> sqrt(16)
4.0
```

On peut aussi importer de cette manière toutes les fonctions d'un module.

```
>>> from math import *
>>> sin(pi/2) + cos(pi/2)
1.0
```

Si vous importez tout un module via `*`, faites attention aux conflits si des fonctions (ou des variables) ont le même nom. La dernière importée ou définie l'emporte !

```
>>> e = 4
>>> f = 5
>>> g = 6
>>> print(e, f, g)
4 5 6
>>> from math import *
>>> print(e, f, g)
2.718281828459045 5 6
```

Documentation sur les modules

Comment connaître les fonctions d'un module ?

Option 1 : Consultez <https://docs.python.org/3/library>

Python » 3.5.2 » Documentation » The Python Standard Library » 9. Numeric and Mathematical Modules » | [previous](#) | [next](#) | [modules](#) | [index](#)

Table Of Contents

- 9.2. **math** — Mathematical functions
 - 9.2.1. Number-theoretic and representation functions
 - 9.2.2. Power and logarithmic functions
 - 9.2.3. Trigonometric functions
 - 9.2.4. Angular conversion
 - 9.2.5. Hyperbolic functions
 - 9.2.6. Special functions
 - 9.2.7. Constants

Previous topic
9.1. **numbers** — Numeric abstract base classes

Next topic
9.3. **cmath** — Mathematical functions for complex numbers

This Page
[Report a Bug](#)
[Show Source](#)

9.2. **math** — Mathematical functions

This module is always available. It provides access to the **mathematical** functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the **cmath** module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much **mathematics** as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

9.2.1. Number-theoretic and representation functions ¶

math.ceil(x)
Return the ceiling of *x*, the smallest integer greater than or equal to *x*. If *x* is not a float, delegates to *x*.**ceil**() , which should return an *Integral* value.

math.copysign(x, y)
Return a float with the magnitude (absolute value) of *x* but the sign of *y*. On platforms that support signed zeros, **copysign**(1.0, -0.0) returns -1.0.

math.fabs(x)
Return the absolute value of *x*.

math.factorial(x)

Option 2 : Utilisez `help()` et `dir()` dans la console interactive

```
>>> import math
>>> help(math)

(longue aide: voir console interactive)

>>> dir(math)

['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh',
'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc',
'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'loglp', 'log2', 'modf',
'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'tau', 'trunc', 'ulp']

>>> help(math.sin)
Help on built-in function sin in module math:

sin(x, /)
    Return the sine of x (measured in radians).
>>> help(math.radians)
Help on built-in function radians in module math:

radians(x, /)
    Convert angle x from degrees to radians.
>>> math.sin(math.radians(45))

0.7071067811865475
```

Remarque : pour le moment nous ignorons la signification du “, /” dans l’aide affichée par `help`, mais nous y reviendrons.

3.2. Définir de nouvelles fonctions

Pourquoi créer des fonctions ?

Diviser un programme en fonctions permet de

- *rassembler et donner un nom* à un groupe d'instructions : programme plus facile à lire et à déboguer
- *éliminer du code répétitif* : programmes plus courts, plus faciles (et moins dangereux) à modifier
- *diviser* des tâches importantes en petits morceaux : permet de déboguer chaque morceau indépendamment
- *réutiliser* du code : des fonctions bien conçues (et souvent silencieuses) sont souvent utiles pour d'autres programmes

Définition de nouvelles fonctions

Au chapitre précédent, nous avons vu comment créer une fonction qui s'utilise ensuite comme les fonctions prédéfinies.

```
import math

def circonference(r):
    return 2 * math.pi * r

print(circonference(10))
print(circonference(2))
```

L'exécution de ce script donnera :

```
62.831853
12.5663706
```

Définition. Une *définition de fonction* spécifie le nom, les arguments (optionnel) et la séquence d'instructions de la fonction.

```
def circonference(r):
    return 2 * math.pi * r

def get_sum(x, y):
    return x + y
```

Une définition de fonction comporte deux éléments : l'*en-tête* et le *corps*.

```
def get_sum(x, y):
    return x + y
```

Spécification de l'*en-tête* de la fonction :

- `def` est un mot-clef qui indique qu'il s'agit d'une définition, il est suivi par le nom de la fonction
- la liste des noms des arguments (séparés par des virgules) est donnée entre parenthèses
- les parenthèses sont obligatoires (même s'il n'y a pas d'argument)
- la liste des arguments est suivie du caractère `:` (deux points)
- les noms des fonctions suivent les mêmes règles que les noms des variables (cf. Chap. 2)

Le nombre d'arguments

- peut être égal à zéro (mais les parenthèses restent nécessaires);
- n'est pas limité.

Exemple : le script suivant

```
def get_hello_word():
    return 'Bonjour'
```



```
def get_name(lastname, firstname, title):
    return title + ' ' + firstname + ' ' + lastname

print(get_hello_word(), get_name('Baroud', 'Bill', 'M.'))
```

affiche

Bonjour M. Bill Baroud

```
def get_sum(x, y):
    return x + y
```

Spécification du *corps* de la fonction :

- les instructions utilisent les arguments comme des variables;
- les instructions doivent être *indentées* : cela permet de grouper les instructions (spécifier ce qui fait partie du corps). Cela fait partie de la *syntaxe* (obligatoire);
- en pratique, l’indentation est un nombre constant de caractères “espace” en début de chaque ligne (*convention* : 4 espaces);
- évitez les tabulations pour indenter (problèmes entre éditeurs);
- on utilise le mot-clef `return` pour retourner une valeur.

Dans le corps d’une fonction,

- le nombre d’instructions n’est pas limité (mais respectez l’indentation);
- d’autres fonctions peuvent être appelées.

Exemple : le script suivant

```
def get_sum(x, y):
    return x + y

def average(a, b, c, d):
    total = 0
    total = get_sum(total, a)
    total = get_sum(total, b)
    total = get_sum(total, c)
    total = get_sum(total, d)
    moyenne = total / 4
    return moyenne

print(average(2, 4, 5, 10))
```

affiche

5.25

La lecture d’une instruction `return` termine l’exécution de la fonction. La valeur qui suit cette instruction est directement retournée et la fonction est interrompue.

Exemple : le script suivant

```
def fonction():
    res = 3
    return res
    res = 5

x = fonction()
print(x)
```

affiche

3

En mode interactif, il faut mettre une ligne blanche pour terminer la définition d’une fonction.

```
>>> def get_sum(x, y):
```

```

    return x + y

>>>

```

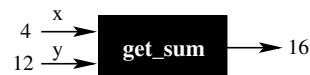
Appel de ses propres fonctions

Une fois définie, une fonction personnelle s'appelle de la même façon que les fonctions prédéfinies. Les appels de fonctions et les expressions peuvent être *composées*.

```

>>> import math
>>> get_sum(4, 12)
16
>>> get_sum(math.pi, math.e)
5.85987448205
>>> get_sum(circonference(10), circonference(2))
75.39822368615503

```



Paramètres

```

>>> def get_sum(x, y):
    return x + y

```

A l'intérieur d'une fonction, les arguments sont appelés des *paramètres*.

- Ils fonctionnent comme des variables (qui sont définies lors de l'appel à la fonction);
- La fonction marche avec n'importe quel type de paramètre sur lequel les instructions sont valides.

```

>>> get_sum('Bon', 'jour')
Bonjour

```

Un argument est évalué avant l'appel à la fonction. Une variable peut être utilisée comme argument : le nom de la variable peut être différent de celui du nom du paramètre. Elle se contente de « transmettre » sa valeur au paramètre.

Illustration : visualisation du script (`arguments.py`) suivant via Python Tutor

```

import math

def circonference(r):
    return 2 * math.pi * r

def get_sum(x, y):
    return x + y

print(get_sum(3, circonference(9 - 4)))
a = 13
b = 5
print(get_sum(a, b))

```

Ordre des arguments

L'ordre des arguments est important!

```

>>> def get_ratio(x, y):
    return x / y

>>> print(get_ratio(12, 4))

```

3.0

```
>>> print(get_ratio(4, 12))
```

0.3333333333333333

Variables locales

Un paramètre est une *variable locale* à sa fonction, c'est à dire qu'il n'existe pas en dehors de sa fonction (en dehors de `get_sum`, `x` et `y` n'existent pas)

```
>>> def get_sum(x, y):
    return x + y

>>> a = get_sum(3, 4)
>>> print(a)
7
>>> print(x)
NameError: name 'x' is not defined
```

Intuition : boîte noire (on ne voit pas le corps de la fonction de l'extérieur de celle-ci)

De la même manière, toute variable définie à l'intérieur de la fonction est une *variable locale*. Elle est détruite quand l'appel à la fonction est terminé.

```
>>> def get_sum(x, y):
    res = x + y
    return res

>>> a = get_sum(3, 4)
>>> print(a)
7
>>> print(res)
NameError: name 'res' is not defined
```

On parle de la *portée* (scope) des variables : zone du programme dans laquelle elle est disponible (*illustration* via Python Tutor ([scope.py](https://pythontutor.com/scope.py)))

3.3. Effets de bord

Effets de bord

Définition. On dit qu'une fonction possède un *effet de bord* si celle-ci produit un effet qui est visible en dehors de la fonction. *Le fait de retourner une valeur n'est pas considéré comme un effet de bord.*

Exemples d'effets de bord :

- afficher quelque chose à l'écran ;
- modifier un fichier ;
- objet passé en argument modifié après l'appel à la fonction¹.

Les fonctions comme `abs` ou `int` ont-elles des effets de bord ?

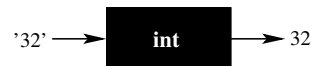
Non. En effet, rien n'est affiché ou visible en dehors de la fonction. La seule chose "visible" (ou plutôt accessible) est la valeur retournée qu'il faut afficher explicitement, si nécessaire :

1. On y reviendra plus tard car pour le moment les types des arguments utilisés (`int`, `str`, etc.) ne sont pas sujets à ce type d'effets de bord.

- soit en utilisant `print` (mode script ou interactif);
- soit en faisant un simple appel à la fonction en mode interactif, ce qui est raccourci déjà vu précédemment qui permet d'éviter d'entrer `print`

```
>>> result = int('32')
>>> print(result)
```

32



La fonction `math.sin` a-t-elle des effets de bord ?

```
>>> math.sin(3)
0.1411200080598672
```

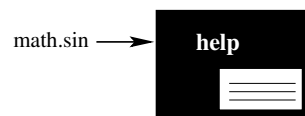
Non. Ici, l’affichage de la valeur retournée est due à l’utilisation du mode interactif. En mode script, ou si on avait assigné la valeur de retour à une variable, rien n’aurait été affiché.

Quizz : quelles fonctions présentées jusqu’ici possèdent des effets de bord ?

Les fonctions `print` et `help` provoquent un affichage. Un comportement de ces fonctions est donc visible en dehors de ce qu’elles pourraient retourner, comme le code suivant le démontre en utilisant une assignation pour éviter l’affichage automatique.

```
>>> x = print(3)
3
>>> x = help(math.sin)
Help on built-in function sin in module math:
sin(...)
    sin(x) -> Return the sine of x (measured in radians).
```

Intuition : un écran d’affichage est apparu sur la boîte noire



Règle générale : les fonctions ne devraient *pas avoir d’effet de bord*, sauf si l’objet de la fonction est explicitement d’en produire un (comme `print`, `help` ou une fonction qui sauvegarderait des données dans un fichier).

Pourquoi ?

Car l’utilisateur d’une fonction sans effet de bord

- peut *contrôler* ce qui, dans son programme, est affiché ou pas ;
- n’a pas le risque de voir des données ou des valeurs *modifiées* sans qu’il le réalise lui même.

Imaginez si `math.sin` affichait intempestivement des choses dans un gros programme de calculs scientifiques !

A contrario, certaines fonctions ont explicitement comme but d’avoir un effet de bord, comme afficher quelque chose.

```
>>> def print_jacques():
    s = 'Frère Jacques'
    print(s + ', ' + s)

>>> print_jacques()
```

Frère Jacques, Frère Jacques

Fonctions sans instruction `return`

```
>>> def print_jacques():  
    s = 'Frère Jacques'  
    print(s + ', ' + s)
```

La fonction précédente a été définie sans instruction `return`, car ce n'était pas son but de retourner quelque chose.

Que se passe-t-il si on essaye néanmoins d'afficher ce qu'elle retourne ?

Valeur de retour `None`

```
>>> x = print_jacques()  
Frère Jacques, Frère Jacques  
>>> print(x)  
None
```

En réalité, en Python, une fonction retourne *toujours* quelque chose mais si son but n'est pas de produire une valeur de retour, alors la valeur spéciale `None` est retournée. C'est la valeur "vide" en Python.

Remarques :

- La valeur `None` est la seule valeur possible d'un type qui lui est dédié, le `NoneType` ;
- Les fonctions `print` et `help` retournent également `None`.

Lors du cours et des TP, nous insisterons très fort, dans le cadre des fonctions, sur le fait que

`return` \neq `print`

La fonction `print` ne doit être utilisée qu'exceptionnellement dans une fonction : uniquement si le rôle de la fonction est d'afficher quelque chose (cela constituera une minorité des fonctions que vous allez définir).

Nous allons illustrer la différence entre `return` et `print` et l'intérêt des fonctions sans effets de bord en créant notre premier module.

Nous créons un module `mymath.py` qui contient le code suivant.

```
def square(x):  
    return x ** 2
```

Il n'y a rien d'autre à faire pour créer un module que de créer un fichier. Nous créons également un script `test.py` qui va importer notre nouveau module.

```
import mymath  
  
x = mymath.square(5)  
print(x)
```

L'exécution du script `test.py` va afficher 25 comme attendu.

Remarque : créer un module en Python est aussi simple que ça ! Plus d'informations sur les modules : voir tutoriel "Modules" sur Moodle.

Dans le scénario précédent, nous avons créé une fonction `square` sans effet de bord grâce à l'utilisation de `return`. C'est dans le script de test que nous avons décidé d'afficher le résultat.

Que ce serait-il passé si nous avions utilisé `print` plutôt que `return` en définissant la fonction

square?

Pour le tester, modifions notre module `mymath.py` avec le code suivant.

```
def square(x):  
    print(x ** 2)
```

Nous ne modifions pas le script `test.py`:

```
import mymath  
  
x = mymath.square(5)  
print(x)
```

Qu'est-ce qui va s'afficher en exécutant `test.py`?

Le résultat affiché est le suivant :

```
25  
None
```

C'est presque le résultat souhaité sauf qu'il y a un `None` qui s'affiche à cause du `print(x)` et du fait que comme il n'y a plus d'instructions `return` dans `square`, cette fonction retourne `None`.

On pourrait être tenté de corriger les choses en supprimant `print(x)` du script `test.py`. Et en apparence, ça fonctionne car maintenant l'affichage est simplement 25.

Cependant, la « correction » précédente est une erreur car nous ne pouvons plus travailler avec la valeur qui est censée être calculée par la fonction `square`. Cela s'observe par exemple avec le nouveau script `test.py` suivant :

```
import mymath  
  
x = mymath.square(5)  
print(x + 5)
```

On aimerait voir 30 s'afficher mais le script produit une exception et un effet de bord, alors que cela aurait parfaitement fonctionné avec la bonne manière de faire (utiliser un `return`).

```
25  
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

3.4. Flot d'exécution

Flot d'exécution

L'ordre dans lequel des instructions sont exécutées est appelé le *flot d'exécution*

- l'exécution commence toujours à la première instruction
- les instructions suivantes sont ensuite exécutées de haut en bas, une après l'autre
- une fonction doit être définie (ou importée depuis un module) *avant* d'être appelée
- les définitions de fonctions n'altèrent pas le flot d'exécution (elles sont lues et deviennent disponibles) *mais* le corps d'une fonction n'est exécuté que quand une fonction est appelée
- un appel à une fonction peut être vu comme un détour dans le programme
- le flot d'exécution n'est donc pas linéaire

Pile de fonctions

Les fonctions qui appellent d'autres fonctions forment une *pile de fonctions* : la dernière fonction appelée (disons f_B) doit être terminée pour "revenir" à la fonction f_A qui contenait l'appel à B .

Exemple : le script suivant est valide et affiche 12. Illustration de la pile via Python Tutor (pile.py)

```
def f_A(x):
    return x + f_B(x)

def f_B(y):
    return y ** 2

print(f_A(3))
```

Pile de fonctions

Si une exception se produit pendant l'appel à une fonction, Python affiche la pile des fonctions en cours.

Exemple : remplaçons le code de `f_B` avec une instruction erronée (`x` n'est pas accessible dans `f_B`)

```
def f_A(x):
    return x + f_B(x)

def f_B(y):
    return x * y

print(f_A(3))
```

Pile de fonctions

L'interpréteur signale l'exception et affiche la pile au moment où celle-ci s'est produite :

```
Traceback (most recent call last):
  File "pile2.py", line 7, in <module>
    print(f_A(3))
  File "pile2.py", line 2, in f_A
    return x + f_B(x)
  File "pile2.py", line 5, in f_B
    return x * y
NameError: name 'x' is not defined
```

Exercice : l'ordre des définitions est-il valide ?

```
def print_jacques():
    print(get_twice('Frere Jacques'))

def get_twice(s):
    return s + ', ' + s

def print_song():
    print_jacques()
    print_dormez()

def print_dormez():
    print(get_twice('Dormez-vous ?'))

print_song()
```

Illustration : via Python Tutor

3.5. Documenter son code

Pour être plus simple à lire, un code source devrait être correctement *documenté*.

On peut ajouter des notes dans du code, qui ne seront pas interprétées. On appelle cela des *commentaires*.

Une manière de faire en Python : tout ce qui suit le symbole # est ignoré jusqu'à la fin de la ligne.

```
# calcule le pourcentage de l'heure écoulée
percent = (minute * 100) / 60
```

```
v = 5    # assigne 5 à v
```

```
v = 5    # vitesse en metres / secondes
```

Quels commentaires trouvez-vous utiles et pourquoi ?

On a vu qu'on peut obtenir de l'aide avec `help`.

```
>>> help(math.sin)
Help on built-in function sin in module math:

sin(...)
    sin(x)

    Return the sine of x (measured in radians).
```

Qu'en est-il de nos fonctions ?

```
>>> help(get_sum)
Help on function get_sum in module __main__:

get_sum(x, y)
```

Un *commentaire multilignes* commence par `"""` et se termine par `"""`.

```
""" this program makes
some useful computation
and is much commented.
"""
x = 2 * 3
y = x + 3 # this is a one line comment
```

Différence entre les commentaires multilignes et les commentaires “fin de ligne” : l'interpréteur ne les ignore pas toujours car ils peuvent être utilisés pour documenter certaines parties du programme, comme une fonction.

Un *docstring* est un commentaire multilignes placé au début du corps d'une fonction.

```
>>> def get_sum(x, y):
    """ return the sum of x and y """
    return x + y

>>> help(get_sum)
Help on function get_sum in module __main__:

get_sum(x, y)
    return the sum of x and y
```

- Documenter clairement ses fonctions : très bonne habitude à prendre !
- Seules les instructions du corps sont cachées, toute l'information utile à l'utilisation d'une fonction est accessible.
- Formatez vos commentaires multilignes pour qu'ils soient facilement lisibles (par ex., max 79 caractères par ligne)

Les *annotations de type* peuvent aussi servir à documenter une fonction, pour avertir l'utilisateur de quels sont les types attendus en entrée et en sortie.

```
def get_sum(x: int, y: int) -> int:
    """ return the sum of x and y """
    return x + y
```

Actuellement, en Python, l'annotation de type ne sert que de documentation pour le programmeur (l'interpréteur ne teste pas si une variable est du type attendu).

Introspection

Les docstring montrent une des formes de l'*introspection* : particularité du langage Python qui lui permet de s'auto-explorer.

```
>>> print(get_sum.__doc__)
return the sum of x and y
>>> print(get_sum.__name__)
get_sum
>>> print(math.sqrt.__module__)
math
>>> print(get_sum.__module__)
__main__
```

Le module `__main__` est le module par défaut.

Objets et introspection

Définir une fonction crée une variable avec le même nom, dont la valeur est un *objet fonction* (de type `function`)

```
>>> print(get_sum)
<function get_sum at 0x10167e048>
>>> print(type(get_sum))
<class 'function'>
```

→ c'est sur (grâce à) cet objet fonction que nous pouvons utiliser l'introspection.

On utilise la notation point pour accéder aux *attributs* d'un objet. La liste des attributs d'un objet est disponible via `dir`.

```
>>> dir(get_sum)
['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get__', '__getattribute__', '__globals__', '__gt__',
 '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__', '__module__', '__name__', '__ne__', '__new__',
 '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>> print(get_sum.__doc__)
return the sum of x and y
```

3.6. Un petit mot sur les Tuples (parenthèse)

En mathématiques, on représente un point dans \mathbb{R}^n comme un n -uplet : collection ordonnée (séquence) de n coordonnées. Par exemple, l'origine du plan est $(0,0)$ et un point dans l'espace peut être représenté par (x,y,z) où x , y et z sont ses coordonnées dans les 3 dimensions de l'espace.

En Python, un *tuple* peut être vu et écrit de manière similaire : séquence de plusieurs valeurs séparées par des virgules et mises entre parenthèses.

```
>>> x = 2
>>> y = 3
>>> point1 = (x, y)
>>> point2 = (4, 5)
>>> type(point1)
<class 'tuple'>
>>> print(point1, point2)
(2, 3) (4, 5)
```

On peut assigner les valeurs d'un tuple dans un autre (possédant le même nombre de valeurs)

```
>>> (a, b) = point1
>>> print(a)
2
>>> print(b)
3
```

Ceci permet de réécrire notre premier algorithme (échanger deux valeurs) très simplement et sans utiliser de variables temporaires :

```
>>> a = 17
>>> b = 21
>>> (b, a) = (a, b)
>>> print(a, b)
21 17
```

Les valeurs d'un tuple ne doivent pas être du même type.

```
>>> matricule = 142543
>>> nom = 'John Doe'
>>> student = (nom, matricule)
>>> print(student)
('John Doe', 142543)
```

Une fonction peut prendre un tuple en paramètre.

```
>>> def sum_pair(t):
    (x, y) = t
    return x + y

>>> sum_pair(point1)
5
```

Une fonction peut retourner un tuple, ce qui lui permet par exemple de retourner plusieurs valeurs :

```
>>> def f(x, y):
    sum = x + y
    prod = x * y
    return sum, prod

>>> s, p = f(3, 4)
>>> print(s, p)
7 12
```

Pour en savoir plus : le chapitre 11 sera consacré au tuples.

3.7. Erreurs fréquentes

Indentation : problèmes quand des espaces et des tabulations sont utilisées (utilisez des espaces exclusivement)

Sauvegarde : n'oubliez pas de sauver votre script avant de l'exécuter, sinon vous ne comprendrez pas pourquoi votre programme ne fonctionne pas (même s'il est correct)

Instruction return : l'instruction `return` termine l'exécution de la fonction, la suite n'est pas exécutée

print : il ne faut pas utiliser `print` au lieu de `return` dans une fonction qui ne doit pas avoir d'effet de bord

3.8. Exercices interactifs

Une fonction est un objet que l'on peut passer en argument à une autre fonction (en omettant les parenthèses). Exemple :

```
def do_twice(f):  
    f()  
    f()
```

```
def print_spam():  
    print('spam')
```

```
do_twice(print_spam)
```

Exercice. Comprenez-vous le code ci-dessus ? Essayez-le en mode script.

Exercice. En éditant le script de l'exercice précédent :

- Modifiez la fonction `do_twice` pour qu'elle prenne deux arguments : une fonction et une valeur. Elle appelle alors la fonction deux fois, en passant la valeur en argument.
- Modifiez la fonction `print_spam` pour qu'elle affiche une chaîne passée en argument.
- Utilisez la version modifiée de `do_twice` pour appeler `print_spam` deux fois, en passant `'hello'` en argument.

3.9. Glossaire

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`).

définition de fonction : instruction qui crée une nouvelle fonction, spécifie son nom, ses paramètres, et les instructions qu'elle doit exécuter.

en-tête : (header) la première ligne de la définition d'une fonction (`def`, nom de la fonction, liste d'arguments, caractère "deux points").

corps : (body) séquence d'instructions dans une définition de fonction.

paramètre : variable utilisée à l'intérieur d'une fonction qui réfère à la valeur passée en argument.

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.

variable locale : variable définie à l'intérieur d'une fonction. Une variable locale ne peut être utilisée qu'à l'intérieur de sa fonction.

portée : 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.

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.

module : fichier qui contient une collection de fonctions et d'autres définitions.

import : instruction qui lit un module et crée un objet module.

__main__ : le module *__main__* est le module par défaut

composition : utiliser une expression comme une partie d'une expression plus grande, ou une instruction comme une partie d'une instruction plus grande.

flot d'exécution : ordre dans lequel les instructions sont exécutées dans un programme.

diagramme de pile : représentation graphique d'une pile de fonctions, leurs variables, et les valeurs qu'elles réfèrent.

traceback : liste de fonctions qui sont exécutées, affichées quand une exception se produit.

commentaire : information dans un programme destinée au lecteur du code source et qui n'a pas d'effet sur l'exécution du programme.

docstring : commentaire multilignes placé au début du corps d'une fonction, destiné à sa documentation.

introspection : particularité du langage Python qui lui permet de s'auto-explorer.

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 créée par une instruction `import` et qui fournit un accès aux valeurs et fonctions définies dans le module.

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.

tuple : en Python, un `tuple` est une séquence de valeurs. Ils permettent par exemple de retourner plusieurs valeurs dans une fonction.

Instructions conditionnelles

Expressions booléennes • Instructions conditionnelles • Fonctions booléennes • Entrées au clavier (parenthèse) • Conversion de réels en entiers (parenthèse) • Erreurs d'approximation (parenthèse) • Erreur fréquente • Exercices interactifs • Glossaire

4.1. Expressions booléennes

Une *expression booléenne* est une expression qui retourne soit vrai, soit faux.

- le type `bool` possède deux valeurs spéciales : `True` (*vrai*) et `False` (*faux*).
- L'opérateur `==` compare deux opérandes et retourne `True` si elles sont égales et `False` sinon.

```
>>> 5 == 5
True
>>> 5 == 6
False
```

Le type `bool`

Les valeurs `True` et `False` sont de type `bool`, ce ne sont pas des `str`.

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
>>> type('False')
<class 'str'>
>>> type(2 == 3)
<class 'bool'>
```

Opérateurs de comparaison

L'opérateur `==` est un *opérateur de comparaison*. Il en existe 5 autres qui retournent tous un `bool` :

- `x != y` *x n'est pas égal à y*
- `x > y` *x est plus grand que y*
- `x < y` *x est plus petit que y*
- `x >= y` *x est plus grand ou égal à y*
- `x <= y` *x est plus petit ou égal à y*

Note : `=<` et `=>` ne sont pas acceptés (erreurs de syntaxe)

Comparaison et assignation : ne pas confondre

Une erreur fréquente est de confondre l'opérateur de comparaison `==` et l'opérateur d'assignation `=`

```
>>> x = 2
>>> type(x == 3)
<class 'bool'>

>>> res = (x == 3)
>>> print(res)
False

>>> type(x = 3)
TypeError: type() takes 1 or 3 arguments

>>> res = (x = 3)
SyntaxError: invalid syntax
```

Opérateurs logiques

Il y a 3 opérateurs logiques : `and`, `or` et `not`.

La sémantique de ces opérateurs est similaire à leur signification en anglais : “et”, “ou” et “non”.

On construit des expressions booléennes en utilisant les opérateurs de comparaison et logiques.

Par exemple,

```
x > 0 and x < 10
```

est vrai si et seulement si `x` est plus grand que 0 *et* plus petit que 10.

Exemples

- `x` est compris entre 2 et 5
`x >= 2 and x <= 5`
- `x` est plus petit que 4 ou plus grand que 10
`x < 4 or x > 10`
- `n` est divisible par 2 ou 3

- ```
n % 2 == 0 or n % 3 == 0
```
- $n$  n'est pas divisible par 4  
`not (n % 4 == 0)`
  - $x$  n'est pas plus grand que  $y$   
`not x > y`

Comment réécrire les deux dernières expressions sans utiliser `not` ?

Remarque : `not` a une précedence plus faible que les opérateurs de comparaisons

### Valeurs non nulles

A priori les opérandes des opérateurs logiques devraient être des `bool` mais Python est moins strict que cela :

```
>>> 17 and True
True
>>> 0 and True
0
```

Toute valeur non nulle est interprétée comme vraie, mais une valeur nulle produit un comportement étrange !

*Attention* : cette flexibilité peut être pratique dans certains cas mais est source de subtilités et confusion : à éviter (sauf si vous savez ce que vous faites).

### Opérateurs de comparaison multiples

Pour tester si  $n$  est compris entre 0 et 9, nous pouvons écrire

```
n >= 0 and n < 10
```

En mathématiques, on écrirait sans doute  $0 \leq n < 10$ .

En Python, l'expression booléenne suivante est également acceptée :

```
0 <= n < 10
```

Cependant, dans la plupart des langages de programmation, utiliser des opérateurs de comparaison multiples est interdit (il faut les composer avec "et") : cette caractéristique de Python est inhabituelle.

### Loi de De Morgan

Les expressions booléennes contenant un `not` appliqué sur une expression composée d'un "et" ou d'un "ou" sont souvent difficiles à comprendre.

```
not (0 < n and n < 1000)
```

*"vrai quand il n'est pas vrai que  $n$  est plus grand que 0 et  $n$  est plus petit que 1000"*

La loi de De Morgan permet de simplifier une expression contenant une négation sur deux termes joints par un "et" ou un "ou" :

$\text{not } (A \text{ and } B)$  est équivalent à  $(\text{not } A) \text{ or } (\text{not } B)$   
 $\text{not } (A \text{ or } B)$  est équivalent à  $(\text{not } A) \text{ and } (\text{not } B)$

Remarquez comme les “et” et “ou” s’inversent.

Ainsi,  $\text{not } (n > 0 \text{ and } n < 1000)$  “vrai quand il n’est pas vrai que  $n$  est plus grand que 0 et  $n$  est plus petit que 1000”

devient

$\text{not } (n > 0) \text{ or } \text{not } (n < 1000)$

puis, après simplification :

$n \leq 0 \text{ or } n \geq 1000$  “vrai quand  $n$  est plus petit ou égal à 0 ou plus grand ou égal à 1000”

Ainsi,  $\text{not } (n \leq 0 \text{ or } n \geq 1000)$  “vrai quand il n’est pas vrai que  $n$  est plus petit ou égal à 0 et  $n$  est plus grand ou égal à 1000”

devient

$\text{not } (n \leq 0) \text{ and } \text{not } (n \geq 1000)$

puis, après simplification :

$n > 0 \text{ and } n < 1000$  “vrai quand  $n$  est plus grand que 0 et plus petit que 1000”

## 4.2. Instructions conditionnelles

### Exécution conditionnelle

Les programmes ont souvent besoin de tester certaines conditions et de modifier leur comportement en fonction de celles-ci.

Les *instructions conditionnelles* permettent de le faire.

*Syntaxe*

`if` expression booléenne :

`instructions`

*Comportement* Les instructions (indentées) sont exécutées si l’expression booléenne retourne vrai. Si ce n’est pas le cas, elles ne sont pas exécutées.

L’expression booléenne après le `if` est appelée la *condition*.

*Exemple*

```
>>> x = 2
>>> if x > 0:
 print('x est positif')

x est positif
>>> if x < 0:
 print('x est negatif')

>>>
```

- Comme pour le corps des fonctions, les instructions se trouvant dans le corps d’un



`if` doivent être indentées

- Il n’y a pas de limite aux nombres d’instructions qui apparaissent dans le corps d’un `if`

### L’instruction `pass`

Dans le corps d’un `if` (et d’une fonction), il doit y avoir au moins une instruction.

Occasionnellement, on a besoin (temporairement) d’un corps qui ne contient pas d’instructions (par ex., pour y ajouter du code plus tard et pouvoir se concentrer d’abord sur le reste).

Dans ce cas : on peut utiliser l’instruction `pass`, qui ne fait rien.

```
def f():
 pass

if x < 0:
 pass # TODO: gerer le cas ou x est negatif!
```

### Exécution alternative

Une deuxième forme d’instruction `if` est l’*exécution alternative*, dans laquelle il y a deux possibilités : la condition détermine laquelle est exécutée.

#### Syntaxe

`if` expression booléenne :

instructions A

else:

instructions B

*Comportement* Les instructions A ne sont exécutées que si l’expression booléenne (la condition) retourne vrai. Si ce n’est pas le cas, les instructions B sont exécutées.

Comme la condition est soit vraie, soit fausse, exactement une des deux alternatives est exécutée. Ces alternatives sont appelées des *branches* car elles permettent de définir différentes branches dans le flot d’exécution.

#### Exemple

```
x = 3

if x % 2 == 0:
 print('x est pair')
else:
 print('x est impair')

x est impair
```

*Attention* : respectez l’indentation. Il faut apprendre à voir comment la gérer correctement avec IDLE, en mode interactif dans une console ou avec un éditeur de texte en mode script.

### Conditions chaînées

Parfois, il y a plus que deux possibilités. Pour créer plus de deux branches on utilise des *conditions chaînées*.

### Syntaxe

if expression booléenne :

    instructions A

elif expression booléenne :

    instructions B

...

else:

    instructions C

**Comportement** Les instructions A ne sont exécutées que si la première condition est vraie. Si ce n'est pas le cas, la deuxième condition est testée et ainsi de suite, jusqu'au `else` qui est exécuté ssi toutes les conditions sont fausses.

### Exemple

```
if x < y:
 print('x est plus petit que y')
elif x > y:
 print('x est plus grand que y')
else:
 print('x et y sont égaux')
```

- `elif` est une abréviation de “else if”
- il n’y a pas de limite au nombre de `elif`
- s’il y a un `else`, il se trouve à la fin, mais ce n’est pas obligatoire (alors, si toutes les conditions sont fausses, il n’y a rien d’exécuté)

```
if x % 2 == 0:
 print('x est pair')
elif x % 3 == 0:
 print('x est divisible par 3')
```

## Conditions imbriquées

Une condition peut être *imbriquée* dans une autre.

```
if x == y:
 print('x et y sont égaux')
else:
 if x < y:
 print('x est plus petit que y')
 else:
 print('x est plus grand que y')
```

La première condition possède deux branches. La première branche contient une instruction seule. La seconde branche contient une autre condition, qui possède également ses deux branches.

Bien que l’indentation montre la structure des conditions imbriquées, elles deviennent vite difficiles à lire. En général, on essaye de les éviter si possible.

Les opérateurs logiques permettent parfois de simplifier les conditions imbriquées.

```
if x > 0:
 if x < 10:
 print('x is a positive single-digit number')
```

La fonction `print` n'est exécutée que si les deux conditions sont vraies. On peut donc écrire :

```
if x > 0 and x < 10:
 print('x is a positive single-digit number')
```

### Valeur de retour et instructions conditionnelles

Pour rappel, l'instruction `return` dans une fonction signifie "termine immédiatement la fonction et utilise l'expression qui suit comme valeur de retour".

```
def valeur_absolue(x):
 if x < 0:
 return -x
 else:
 return x
```

Si une fonction qui doit retourner une valeur contient des instructions conditionnelles, il faut s'assurer que toutes les possibilités rencontrent une instruction `return`.

```
>>> def valeur_absolue(x):
 if x < 0:
 return -x
 elif x > 0:
 return x

>>> print(valeur_absolue(0))
```

`None`

Une instruction `return` qui n'est pas suivie d'une expression peut être utilisée pour sortir d'une fonction prématurément. Dans ce cas, la valeur spéciale `None` est automatiquement retournée.

```
def aire(r):
 if r < 0:
 return
 return math.pi * r ** 2
```

## 4.3. Fonctions booléennes

Les fonctions peuvent retourner des booléens, ce qui est souvent très utile pour cacher des tests compliqués dans des conditions ou rendre le code plus clair.

*Exemple :*

```
>>> def is_divisible(x,y):
 if x % y == 0:
 return True
 else:
 return False

>>> is_divisible(6,4)

False

>>> is_divisible(6,3)
```

True

C'est une bonne habitude de nommer les fonctions booléennes comme des questions dont la réponse est "oui" (True) ou "non" (False) : `est_pair`, `est_premier`, `is_positive`.

Les fonctions booléennes augmentent la lisibilité des instructions conditionnelles.

```
if is_divisible(x,2) and is_positive(x):
 print('x est un nombre pair positif')
```

Comparer explicitement la valeur de retour d'une fonction booléenne est inutile et alourdit le code.

```
if is_divisible(x,y) == True:
 print('x est divisible par y')
```

Comme un opérateur de comparaison retourne une valeur booléenne, on peut écrire `is_divisible` de manière plus concise :

```
def is_divisible(x,y):
 return x % y == 0
```

### Evaluation "lazy"

Lors d'un test `and` ou `or`, Python effectue une évaluation *lazy* :

- A `and` B : si A est faux, le test B n'est pas évalué, retourne False
- A `or` B : si A est vrai, le test B n'est pas évalué, retourne True

Utile si le test B est par exemple un appel à une fonction booléenne qui prend du temps à être calculée.

*Illustration* : script `lazy.py` présenté en auditoire.

## 4.4. Entrées au clavier (parenthèse)

La fonction `input` permet d'obtenir des données entrées au clavier par l'utilisateur.

- quand elle est appelée, le programme s'arrête et attend que l'utilisateur entre quelque chose au clavier
- quand l'utilisateur appuie sur Return ou Enter, le programme reprend et `input` retourne ce que l'utilisateur a entré sous la forme d'un `str`

```
>>> data = input()
qu'attends-tu?
>>> print(data)
```

qu'attends-tu?

*Remarque* : en Python 2, on utilise `raw_input` au lieu d'`input`

Avant d'attendre une entrée de l'utilisateur, il est utile de lui préciser ce qu'on attend de lui ! La fonction `input` prend un argument de type `str` pour ce faire.

```
>>> name = input('Quel est votre nom ? ')
Quel est votre nom ? John Doe
>>> print(name)
```

John Doe

Le caractère spécial `\n` peut être utilisé dans un `str` et représente une nouvelle ligne.

```
>>> name = input('Quel est votre nom ?\n')
```

Quel est votre nom ?

John Doe

```
>>> print(name)
```

John Doe

Si vous voulez obtenir un nombre entier ou réel, il faut convertir le `str` retourné par `input`.

```
>>> prompt = 'Quel est votre age ?\n'
```

```
>>> data = input(prompt)
```

Quel est votre age ?

19

```
>>> print(data, type(data))
```

```
19 <class 'str'>
```

```
>>> age = int(data)
```

```
>>> print('Vous allez bientôt avoir', (age + 1), 'ans')
```

Vous allez bientôt avoir 20 ans

Si l'utilisateur entre quelque chose qui ne peut être correctement converti, on obtient une erreur.

```
>>> prompt = 'Quel est votre taille ?\n'
```

```
>>> data = input(prompt)
```

Quel est votre taille ?

Heu.... en centimetres ou en metres ?

```
>>> taille = float(data)
```

```
ValueError: invalid literal for float()
```

*Rappel* : Comment appelle-t-on ce type d'erreurs ?

On verra comment gérer cela plus tard.

### Interaction avec l'utilisateur : exemple

*Problème.* Ecrire un programme qui permette de calculer l'aire ou la circonférence d'un cercle de rayon  $r$ .

Une solution :

```
import math
```

```
def aire(r):
```

```
 return math.pi * r ** 2
```

```
def circonference(r):
```

```
 return math.pi * 2.0 * r
```

```
def print_res(res,type):
```

```
 print(type, '=', res)
```

```
prompt = 'Veuillez entrer le rayon du cercle...\n'
```

```
rayon = float(input(prompt))
```

```
if rayon < 0.0:
```

```

 print('Je vais avoir du mal a calculer ca...')
else:
 prompt = 'Voulez-vous connaitre l\'aire ou la circonference ?\n'
 typeCalcul = input(prompt)
 if typeCalcul == 'aire' or typeCalcul == 'Aire':
 res = aire(rayon)
 print_res(res, typeCalcul)
 elif typeCalcul == 'circonference' or typeCalcul == 'Circonference':
 res = circonference(rayon)
 print_res(res, typeCalcul)
 else:
 print('Je ne comprend que \"aire\" ou \"circonference\"')

```

## 4.5. Conversion de réels en entiers (parenthèse)

Pour rappel, si  $x$  et  $y$  sont deux entiers positifs, le résultat de  $x // y$  (division entière) est la partie entière de  $\frac{x}{y}$ .

D'autre part, la fonction `int` avec un nombre réel  $x$  en argument retourne la partie entière de  $x$ .

```

>>> 1 // 2
0
>>> int(0.5)
0

```

*Attention :* en réalité, la division entière calcule un *plancher*<sup>1</sup>, alors que `int` *tronque* la partie décimale. Ainsi, avec des nombres négatifs, les comportements sont différents.

```

>>> -1 // 2
-1
>>> int(-0.5)
0

```

Sachant que `int` tronque un réel, les fonctions suivantes peuvent être utiles pour convertir un réel en entier.

```

>>> import math
>>> x = 1.34
>>> y = -3.29
>>> z = 2.0

```

- **Plancher** (plancher de  $x$  = plus grand entier  $i$  tel que  $i \leq x$ )

```

>>> print(math.floor(x), math.floor(y), math.floor(z))
1 -4 2

```

- **Plafond** (plafond de  $x$  = plus petit entier  $i$  tel que  $i \geq x$ )

```

>>> print(math.ceil(x), math.ceil(y), math.floor(z))
2 -3 2

```

- **Arrondi** (entier le plus proche)

```

>>> print(round(x), round(y), round(z))
1 -3 2

```

*Remarque :* en Python 2, les fonctions d'arrondi retournent des réels et non des entiers, qu'il faut alors convertir en entier.

En ce qui concerne l'arrondi, en Python 3, un nombre positif dont la partie entière est impaire se terminant par 0.5 est arrondi vers le haut. C'est l'inverse si la partie entière est paire.

```

>>> print(round(1.5), round(2.5), round(-1.5), round(-2.5))
2 2 -2 -2

```

---

1. le plancher de  $x$  est le plus grand entier  $i$  tel que  $i \leq x$ .

Les choses s'inversent encore si le nombre est négatif.

Voyez-vous pourquoi un "demi" nombre n'est pas toujours arrondi vers le haut (comme c'est le cas en Python 2)?

Le but est d'éviter un *biais* s'il y a beaucoup d'arrondis, toujours vers le haut, par exemple dans des applications financières.

## 4.6. Erreurs d'approximation (parenthèse)

### Tester l'égalité de deux `float`

Quand on veut tester l'égalité de deux `float`, il faut éviter d'utiliser `==`.

```
>>> x = 0.1
>>> y = (math.sqrt(math.sqrt(x))) ** 4
>>> x == y
False
```

Tout calcul impliquant un `float` peut provoquer une *erreur d'approximation*.

### Erreurs d'approximation

```
>>> print(x, y)
0.1 0.09999999999999999
```

`float` = erreurs d'approximation

En réalité, même pour `x` il y avait déjà une erreur! On peut contrôler le nombre de décimales affichées pour un `float` :

```
>>> print(x)
0.1
>>> print('%3f' % x)
0.100
>>> print('%50f' % x)
0.100000000000000000555111512312578270211815834045410
>>> print('%3f' % y)
0.100
>>> print '%50f' % y
0.09999999999999999167332731531132594682276248931885
```

### Solution

Pour tester l'égalité de deux `float`, il faut vérifier qu'ils sont suffisamment proches, à  $\epsilon = 10^{-9}$  près par exemple.

Ceci peut être écrit grâce à une petite fonction booléenne.

```
>>> def almost_equals(x, y):
 return abs(x - y) < 10e-9

>>> x == y
False
>>> almost_equals(x, y)
True
```

*Remarque :* notez que vous pouvez utiliser une notation scientifique pour les `float`, comme illustré ci-dessus.

## Représentation binaire

Certaines erreurs d'approximation proviennent de la manière dont les nombres sont représentés en machine.

Les chiffres utilisés en base 10 vont de 0 à 9, en base 2, nous n'avons besoin que de 0 et de 1 (binaire). Les nombres sont représentés en base 2 sur un ordinateur.

Pourquoi ? Processeurs composés de millions de transistors (imprimés sur un circuit électronique) : chacun ne gère qu'un bit : 0 (le courant ne passe pas) et 1 (le courant passe).

Plus de détails : voir cours *Fonctionnement des ordinateurs*

*Exemple :* le nombre 35 est écrit

- 35 en base 10 car

$$3 \times 10^1 + 5 \times 10^0 = 30 + 5 = 35;$$

- 100011 en base 2 car

$$1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 32 + 2 + 1 = 35.$$

Il en va de même pour les fractions.

*Exemple :* la fraction  $\frac{1}{4}$  est écrite

- 0.25 en base 10 car

$$\frac{2}{10^1} + \frac{5}{10^2} = \frac{2}{10} + \frac{5}{100} = \frac{25}{100} = \frac{1}{4};$$

- 0.01 en base 2 car

$$\frac{0}{2^1} + \frac{1}{2^2} = \frac{1}{4}.$$

## Erreurs d'approximation

En base 10, la fraction  $\frac{1}{3}$  ne peut être représentée de manière exacte, on doit l'approximer car le terme 3 se répète de manière infinie :

$$0.3333333333333333 \dots$$

Par contre, la fraction  $\frac{1}{10}$  peut être représentée de manière exacte : 0.1

Cependant, en base 2, la fraction  $\frac{1}{10}$  est représentée par :

$$0.000110011001100110011 \dots$$

où le terme 0011 se répète de manière infinie.

Le nombre de bits utilisés par l'ordinateur pour représenter un nombre étant fini explique pourquoi on obtient :

```
>>> print('% .20f' % 0.1)
0.10000000000000000555
```

*Remarques :*

- ce n'est donc pas un "bug" de Python (en réalité Python n'y est pour rien)
- l'erreur peut varier d'une machine à l'autre en fonction du nombre de bits alloués pour y représenter un nombre



## 4.7. Erreur fréquente

### Effet de bord

Comme expliqué au chapitre précédent, la majorité de vos fonctions seront des fonctions *sans effet de bord*. C'est le cas par exemple de la fonction `circonference` ci-dessous.

```
import math

def circonference(rayon):
 return 2 * math.pi * rayon
```

Mathématiquement, un cercle ne peut pas avoir un rayon négatif. On pourrait donc ajouter un test au début de la fonction pour le vérifier.

Pour simplifier, on va supposer que le rayon est bien un nombre : il reste à tester qu'il ne soit pas négatif.

Comment mettriez-vous cela en place sachant qu'on ne veut pas d'effet de bord ?

*Mauvaise solution :*

```
import math

def circonference(rayon):
 if rayon < 0:
 print('Erreur: rayon doit etre >= 0')
 else:
 return 2 * math.pi * rayon
```

Problème : dans certains cas, cette fonction possède maintenant un effet de bord !

```
>>> x = circonference(-10)
Erreur: rayon doit etre >= 0
>>> print(x)
None
```

*Solution sans effet de bord :*

- retourner une valeur utilisée comme *code d'erreur* (-1, None, ...)
- préciser dans la documentation ce que signifie ce code d'erreur

```
def circonference(r):
 """ Retourne la circonference d'un cercle de rayon r
 r -- rayon du cercle >=0 (retourne None si r < 0)
 """
 if r < 0:
 return None
 else:
 return 2 * math.pi * r
```

*Avantage :* contrôle de la gestion de l'erreur *en dehors* de la fonction

```
x = circonference(-10)
if x != None:
 print('x =', x)
else:
 print('Entrez un nombre >= 0 pour le rayon')
```

## 4.8. Exercices interactifs

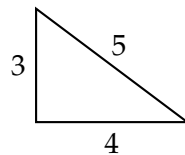
Le dernier théorème de Fermat<sup>2</sup> affirme qu'il n'y a pas d'entiers positifs  $a$ ,  $b$  et  $c$  tels que

$$a^n + b^n = c^n,$$

pour n'importe quelle valeur de  $n$  plus grande que 2.

*Exercice.* Ecrivez une fonction booléenne `check_fermat` qui retourne vrai ssi  $a^n + b^n = c^n$  et exploitez cette fonction pour faire un programme permettant à l'utilisateur de tenter de trouver un contre-exemple au théorème.

Avec 3 bâtons, vous pouvez (ou non) les arranger pour former un triangle (non dégénéré). Etant données 3 longueurs, si l'une d'entre elles est strictement plus grande que la somme des deux autres, alors c'est impossible de former un triangle. Sinon, c'est possible.



*Exercice.* Ecrivez une fonction booléenne `is_triangle` qui, à partir de 3 entiers correspondant à des longueurs, retourne vrai ssi on peut former un triangle avec ces longueurs. Utilisez cette fonction pour faire un programme permettant à l'utilisateur de savoir si 3 longueurs peuvent correspondre à un triangle.

## 4.9. Glossaire

*expression booléenne* : expression qui retourne soit vrai (True), soit faux (False)

*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`.

*instruction conditionnelle* : instruction qui contrôle le flot d'exécution en fonction de certaines conditions

*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

---

2. Enoncé au 17ième siècle dans une marge et prouvé par Andrew Wiles en 1994.

## Réversivité

*Réversivité • Un langage complet (parenthèse) • Quelques algorithmes réversifs • Paramètres par défaut et arguments mots-clefs (parenthèse) • Erreurs fréquentes et debug • Exercice interactif • Glossaire*

---

### 5.1. Réversivité

#### Illustration de la réversivité

*Illustration en auditoire* : énigme présentée en auditoire et résolue par un script dont le code sera dévoilé plus tard.

#### Preuve par récurrence

La réversivité est une notion classique en mathématiques quand il s'agit de prouver une assertion (preuve par récurrence) : cf. cours *Mathématiques élémentaires*

*Rappel du principe*

A prouver : assertion  $S(n) = f(n) \quad \forall n \geq n_0$

*Base* : preuve de l'assertion pour certaines petites valeurs de  $n$  ( $n_0$  par exemple)

*Etape de récurrence* : on suppose que c'est vrai pour  $n \leq k$  (avec  $k \geq n_0$ ), prouver que c'est également vrai pour  $n = k + 1$

*Remarque* : on peut aussi supposer que c'est vrai pour  $n \leq k - 1$  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.

*Exemple*

Soit  $S(n) = \sum_{i=1}^n i$ . Prouver que

$$S(n) = \frac{n(n+1)}{2} \quad \forall n \geq 1 \tag{5.1}$$

Base : si  $n = 1$ , alors  $S(1) = 1$  par définition et (5.1) devient

$$1 = \frac{1 \times 2}{2},$$

ce qui est exact.

Étape de récurrence : on suppose que (5.1) est vraie pour  $1 \leq n \leq k-1$ , prouvons que c'est également vrai pour  $n = k$ . Par hypothèse de récurrence,

$$S(k-1) = \frac{(k-1)k}{2} \quad (5.2)$$

Comment exprimer  $S(k)$  en fonction de  $S(k-1)$ ? Par définition,

$$S(k) = \sum_{i=1}^k i = \sum_{i=1}^{k-1} i + k = S(k-1) + k.$$

Par (5.2), il s'ensuit

$$S(k) = \frac{(k-1)k}{2} + k.$$

Or,

$$\frac{(k-1)k}{2} + k = \frac{k^2 - k}{2} + \frac{2k}{2} = \frac{k^2 + k}{2} = \frac{k(k+1)}{2}.$$

On a donc bien

$$S(n) = \frac{n(n+1)}{2} \quad \forall n \geq 1.$$

Qu'avons-nous du déterminer pour utiliser une preuve par récurrence?

- (base) résoudre un cas simple
- (étape de récurrence)
  - supposer le résultat vrai pour  $n \leq k-1$
  - exprimer  $S(k)$  en fonction de  $S(k-1)$  (ou en fonction d'autres valeurs plus petites que  $k-1$ )
  - retrouver le résultat en combinant les deux points ci-dessus

## Réversivité

En informatique, on peut également utiliser la *réversivité*

*Problème.* Comment calculer la somme des  $n$  premiers nombres entiers?

*Solution 1 :* utiliser la formule prouvée précédemment (formule d'Euler)

```
>>> def sum_1_to_n(n):
 return n * (n + 1) / 2

>>> print(sum_1_to_n(10))
```

55

*Solution 2 :* utiliser la récursivité, c'est-à-dire la *possibilité pour une fonction de s'appeler elle-même*.

- (base) Si  $n = 1$ , alors la somme  $S(n)$  vaut 1
- (étape de récurrence) Si  $n > 1$ ,  $S(n) = S(n-1) + n$

Ces deux éléments suffisent pour résoudre le problème de manière récursive

```
>>> def sum_rec(n):
 if n == 1:
 return 1
 else:
 return sum_rec(n - 1) + n

>>> print(sum_rec(10))
```

55

*Intuition* : le programmeur paresseux

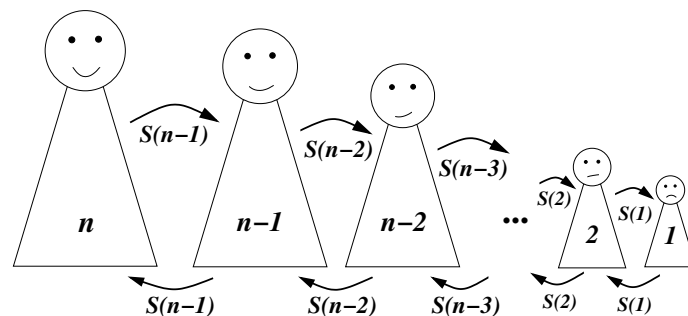
Supposons que Bill soit paresseux et qu'il doive résoudre un problème complexe. Il préfère laisser à Bob le soin de faire le gros du travail. Si Bill est capable de

- résoudre le problème pour les cas les plus simples;
- récupérer le travail de Bob et réaliser un travail de taille un peu plus grande à partir de celui-ci;

alors le problème peut être résolu par récursivité, de manière simple.

*Exemple* : Bill doit calculer  $S(n) = \sum_{i=1}^n i$ . Bill peut demander à Bob de calculer  $S(k)$  pour  $k = 2, 3, \dots, n-1$ . Il lui reste simplement à

- calculer  $S(1)$ ,
- déterminer comment calculer  $S(n)$  en utilisant  $S(n-1)$  (ou d'autres valeurs calculées par Bob).



*Illustration* : appel à `sum_rec(4)` dans Python Tutor

*Problème*. Ecrire une fonction qui calcule la factorielle d'un nombre entier.

Une solution récursive :

- (base)  $0! = 1$
- (étape de récurrence)  $n! = n \times (n-1)!$

En Python :

```
>>> def factorial(n):
 if n == 0:
 return 1
 else:
 recurse = factorial(n - 1)
 result = n * recurse
 return result

>>> print(factorial(4))
```

24

*Problème*. Ecrire une fonction qui calcule  $a^n$ , si  $n \geq 1$ .

Une solution récursive :

- (base)  $a^1 = a$
- (étape de récurrence)  $a^n = a^{n-1} \times a$

En Python :

```
>>> def expo(a, n):
 if n == 1:
 return a
 else:
 return expo(a, n - 1) * a

>>> print(expo(3,3), (3**3))
```

27 27

## Rappels concernant les fonctions

Au chapitre 3, nous avons vu que :

- une fonction peut être utilisée dans une expression comme une valeur (= sa valeur de retour)
- une fonction peut en appeler une autre

La récursivité n'est rien d'autre que l'application de ses deux faits sur la fonction elle-même !

```
def expo(a, n):
 if n == 1:
 return a
 else:
 return expo(a, n - 1) * a
```

Pour comprendre et valider une fonction récursive, il s'agit

- d'accepter qu'un appel récursif retourne la bonne valeur
- de s'assurer que le(s) cas de base(s) soi(en)t bien géré(s)
- de s'assurer que chaque appel récursif réduise la taille du problème jusqu'à atteindre un cas de base

Les exemples précédents montrent que l'on peut résoudre très simplement certains problèmes si on arrive à :

- résoudre le problème pour les cas les plus simples ;
- récupérer une solution pour une certaine taille du problème et construire une solution de plus grande taille à partir de celui-ci.

Dans beaucoup de cas, ces deux tâches sont plus faciles à résoudre qu'une solution "générale" ou "complète" au problème !

Mais... cela peut paraître "magique" : on peut avoir du mal à "accepter" que cela fonctionne aussi simplement.

On peut analyser le flot d'exécution pour s'en convaincre.

## Récursivité et flot d'exécution

```
>>> def factorial(n):
 if n == 0:
 return 1
 else:
 recurse = factorial(n - 1)
 result = n * recurse
 return result

>>> print(factorial(3))
```

6

⇒ analyse du flot d'exécution via Python Tutor

*Problème.* Soit la fonction récursive suivante, comment prédire ce qu'elle va faire? Essayez de décrire le flot d'exécution après un appel à `post_count(3)`.

```
>>> def post_count(n):
 if n <= 0:
 print('Boum')
 else:
 print(n)
 post_count(n - 1)
>>> post_count(3)
>>> post_count(3)
3
2
1
Boum
```

⇒ analyse du flot d'exécution via Python Tutor

*Problème.* Que se passe-t-il si on modifie légèrement la fonction précédente en déplaçant l'appel récursif *avant* l'instruction "`print(n)`"?

```
>>> def pre_count(n):
 if n <= 0:
 print('Boum')
 else:
 pre_count(n - 1)
 print(n)
>>> pre_count(3)
```

Pour résoudre ce problème, on peut refaire l'analyse du flot détaillée, mais ce n'est pas nécessaire si l'on "accepte" que la récursivité fonctionne et qu'on lit le corps du `else` comme suit :

```
Etape de recurrence:
 pre_count(n - 1)
 print(n)
```

Ce qui s'interprète par

- on réalise d'abord le travail pour les valeurs plus petites que  $n$
- ensuite on affiche  $n$

```
>>> pre_count(3)
Boum
1
2
3
```

## Définitions récursives

La récursivité est très naturelle pour résoudre des problèmes qui sont définis récursivement.

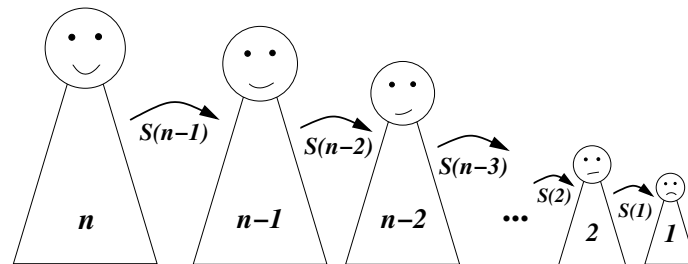
Par exemple, la séquence des nombres de Fibonacci est définie mathématiquement de manière récursive :

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_n &= F_{n-1} + F_{n-2}, \quad \forall n \geq 2. \end{aligned}$$

En Python :

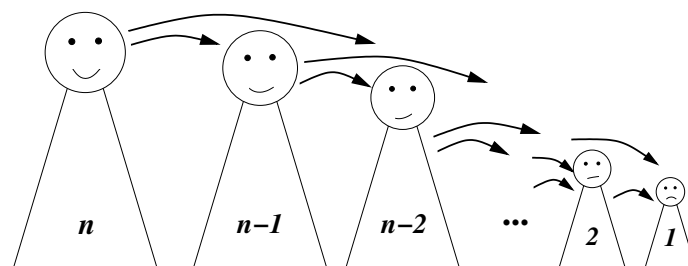
```
def fibo(n):
 if n == 0:
 return 0
 elif n == 1:
 return 1
 else:
 return fibo(n - 1) + fibo(n - 2)
```

### Réurrences faible et forte



*Récurrence faible* : n'utilise que la valeur précédente ( $n - 1$ ).

*Exemples* : factorielle, exposant, etc. vus ci-avant.



*Récurrence forte* : utilise plusieurs (voire toutes les) valeurs précédentes ( $\leq n - 1$ ).

*Exemple* : Fibonacci qui utilise les valeurs  $F(n - 1)$  et  $F(n - 2)$ .

### Récursion infinie

Si une récursion n'atteint jamais un cas de base, le programme ne se termine (théoriquement) jamais ! C'est une *récursion infinie*.

```
def recurse():
 recurse()
```

→ voir la console interactive pour le comportement de cette fonction

Important de considérer les cas de base *et* le fait que chaque appel récursif doit s'approcher d'un cas de base.

En réalité, la récursion infinie n'existe (heureusement) pas, car une exception est lancée



quand le nombre d'appels récursif dépasse une certaine limite. On peut la connaître (et la modifier) grâce à certaines fonctions du module `sys`.

```
>>> import sys
>>> print(sys.getrecursionlimit())
```

```
1000
```

```
>>> sys.setrecursionlimit(10000)
```

Si cette limite n'était pas définie, le programme "planterait" une fois que la mémoire allouée sur la machine pour appeler les milliers d'appels récursifs serait dépassée, ce qui serait bien plus gênant qu'une exception (gérable, voir plus tard).

Que se passe-t-il si on appelle `factorial` avec 1.5 comme argument?

```
>>> sys.setrecursionlimit(10000)
>>> factorial(1.5)
```

```
... RuntimeError: maximum recursion depth exceeded
```

La base ( $n == 0$ ) n'est jamais atteinte :  $1.5 \rightarrow 0.5 \rightarrow -0.5 \rightarrow -1.5 \rightarrow \dots$

## Tester les types

Pour éviter ce genre de problèmes, on peut utiliser la fonction booléenne `isinstance` qui vérifie si un argument est d'un type donné.

```
>>> def factorial(n):
 if not isinstance(n, int):
 print('Factorial is only defined for integers.')
 return None
 elif n < 0:
 print('Factorial is only defined for positive integers.')
 return None
 elif n == 0:
 return 1
 else:
 return n * factorial(n - 1)
```

```
>>> factorial('fred')
Factorial is only defined for integers.
None
>>> factorial(-2)
Factorial is only defined for positive integers.
None
```

## 5.2. Un langage complet (parenthèse)

Jusqu'à présent nous n'avons couvert qu'une petite partie de Python, mais il s'agit pourtant déjà d'un *langage complet*, c.-à-d. que *tout* problème qui peut être résolu de manière algorithmique peut être exprimé dans ce langage!

- tout programme déjà écrit pourrait être réécrit en utilisant uniquement les caractéristiques du langage vues\*
- le reste du langage permet de faire les choses plus facilement
- prouver cette affirmation n'est pas facile, mais cela a été fait par Alan Turing (mathématicien, 1912 – 1954, souvent considéré comme le père de l'informatique et qui a donné son nom à une de vos salles machines)



\* En réalité, vous auriez besoin de quelques commandes supplémentaires pour contrôler des éléments comme la souris, les fichiers, etc.

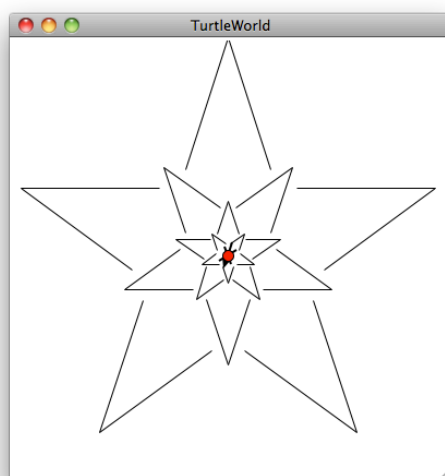
## 5.3. Quelques algorithmes récursifs

Nous allons illustrer / utiliser la récursivité

- en étudiant le comportement récursif de notre tortue (voir tutoriel “UTurtle”);
- en concevant un algorithme récursif pour calculer (une approximation de) la racine carrée d’un nombre.
- en dévoilant le code qui a permis de résoudre l’énigme.

### Une tortue récursive

Réalisation en auditoire du Tutoriel “UTurtle” et illustration de la récursivité.



### La méthode de Héron

Pour approximer une racine carrée, on utilisera la méthode de Héron qui peut se résumer en une phrase :

*Si  $x$  est une approximation de  $\sqrt{a}$ , alors  $\frac{x+a/x}{2}$  est une meilleure approximation de  $\sqrt{a}$ .*

*Exercice.* Ecrire une fonction qui calcule une approximation de la racine carrée d’un nombre réel.

**Etape 1 : quelques essais dans la console interactive**

Dans la console interactive : test “à la main” de la méthode de Héron pour calculer  $\sqrt{4}$ .

Première approximation utilisée : 1

```
>>> a = 4
>>> x = 1
>>> (x + a / x) / 2.0
2.5
>>> x = 2.5
>>> (x + a / x) / 2.0
2.05
>>> x = 2.05
>>> (x + a / x) / 2.0
2.000609756097561
```

**Etape 2 : spécification de la fonction `racine`**

On va écrire une fonction `racine` récursive pour appliquer la méthode de Héron.

*Entrée* : deux nombres réels  $a$  et  $x$

*Sortie* : un nombre réel qui est une meilleure approximation de  $\sqrt{a}$  que  $x$

```
def racine(a, x):
 pass # TODO
```

**Etape 3 : cas de base**

Le *cas de base* de la fonction `racine` équivaut au fait que l’approximation n’est plus significativement meilleure (car la méthode converge)

→ on s’arrête quand la nouvelle approximation est “égale” à la précédente (i.e., suffisamment proche, cf. chap. 4)

Il ne faut pas oublier de retourner cette approximation finale pour permettre que cette information “remonte”.

Comme expliqué au Chap. 4, on ne teste pas si 2 valeurs réelles sont égales mais on vérifie qu’elles soient suffisamment proches.

```
>>> import math
>>> def est_egal(x, y):
 eps = 10**-9
 return abs(x - y) < eps

>>> x = 0.0
>>> y = math.cos(math.pi / 2.0)
>>> x == y
False
>>> est_egal(x, y)
True
>>> print(x, y)
0.0 6.123233995736766e-17

def racine(a, x):
 new = (x + a / x) / 2.0
 if est_egal(x, new):
 return x
```

```

else:
 pass # cas récursif TODO

```

#### Etape 4 : cas récursif

Le *cas récursif* devient facile : il sert à faire “remonter” l’approximation finale en appliquant littéralement la méthode de Héron.

```

def racine(a, x):
 new = (x + a / x) / 2.0
 if est_egal(x, new):
 return new
 else:
 return racine(a, new)

```

⇒ démonstration avec *Python Tutor* quand on oublie le `return`

#### Etape 5 : fonction “wrapper”

La fonction récursive `racine` nécessite deux paramètres en entrée, alors que seul  $a$  est nécessaire pour calculer  $\sqrt{a}$  d’un point de vue “utilisateur”.

On peut “emballer” (to wrap) le premier appel récursif dans une fonction qui initialisera les paramètres supplémentaires (ici, la première approximation).

```

>>> def racine_carree(a):
 return racine(a, 1)

>>> racine_carree(4)
2.0
>>> est_egal(racine_carree(4), 2)
True

```

Une autre solution : utiliser un paramètre par défaut (voir parenthèse ci-après).

#### Résolution de l’énigme

*Illustration* : présentation du code de l’énigme résolue en début de séance.

## 5.4. Paramètres par défaut et arguments mots-clefs (parenthèse)

### Paramètres par défaut

Un *paramètre par défaut* est une valeur par défaut que l’on assigne à un paramètre lors de la définition d’une fonction. Cela permet de rendre un paramètre optionnel.

```

>>> def racine(n, x = 1):
 ...

>>> racine(4)
2.0

```

Les paramètres optionnels doivent se trouver après les paramètres obligatoires. Avec les fonctions récursives, ils sont utiles pour ne pas devoir écrire de fonctions “wrapper”.

### Arguments “mots-clefs”

Un *argument mot-clef* est un argument nommé lors d’un appel à une fonction. Cela permet de modifier l’ordre des arguments, ce qui peut être utile avec des arguments optionnels.

```
>>> def f(x, y = 2, z = 3):
 print('x = ' + str(x) + ', y = ' + str(y) + ', z = ' + str(z))

>>> f(6, 5, 7)
x = 6, y = 5, z = 7
>>> f(7)
x = 7, y = 2, z = 3
>>> f(9, 10)
x = 9, y = 10, z = 3
>>> f(7, z = 8)
x = 7, y = 2, z = 8
>>> f(y = 2, z = 1)
TypeError: f() takes at least 1 non-keyword argument (0 given)
```

Vous pouvez imposer que certains arguments soient obligatoirement *positionnés* et ne peuvent dès lors pas être utilisés comme arguments mots-clefs en les plaçant devant un « , / »

```
>>> def f(x, y = 2, /, z = 3):
 print('x = ' + str(x) + ', y = ' + str(y) + ', z = ' + str(z))

>>> f(7, z = 8)
x = 7, y = 2, z = 8
>>> f(7, y = 8)
TypeError: f() got some positional-only arguments passed as keyword arguments: 'y'
```

*Remarque* : c’est ce que nous avons parfois vu précédemment dans l’aide de certaines fonctions.

## 5.5. Erreurs fréquentes et debug

Pour les erreurs de syntaxe et les exceptions, Python affiche un message : l’information la plus utile dans ce message est :

- le type d’erreur (dernière ligne) : `SyntaxError`, `RuntimeError`, ...
- l’endroit où cette erreur apparaît.

Cependant, il n’est pas toujours facile de trouver l’endroit *exact* du code qu’il faut corriger avec ces informations.

Erreur d’indentation :

```
>>> x = 5
>>> y = 6
File "<stdin>", line 1
 y = 6
 ^
IndentationError: unexpected indent
```

Exceptions sur les valeurs dans des opérations mathématiques (ici, dues à la division entière qui n’aurait pas dû être utilisée)

```
>>> import math
>>> signal = 9
```

```
>>> noise = 10
>>> ratio = signal // noise
>>> decibels = 10 * math.log10(ratio)
File "<stdin>", line 1, in <module>
 decibels = 10 * math.log10(ratio)
ValueError: math domain error
>>> invert = 1 / ratio
File "<stdin>", line 1, in <module>
 invert = 1 / ratio
ZeroDivisionError: integer division by zero
```

Ici : le message n'indique pas où se trouve le problème (qui devrait être corrigé lors du calcul de `ratio`)

Si on obtient une exception ou une erreur sémantique lors de l'appel d'une fonction, il y a 3 possibilités :

- problème au niveau des arguments passés à la fonction : une *précondition*<sup>1</sup> est violée (par exemple un argument dont le type n'est pas géré par la fonction, ou une valeur négative pour un calcul sur des nombres positifs)
- problème au niveau du code de la fonction (corps) : une *postcondition*<sup>2</sup> est violée (par exemple, une erreur sémantique dans le corps de la fonction qui produit un mauvais résultat)
- problème au niveau de la valeur de retour (est-elle correctement utilisée?)

Déboguer une fonction "à la main" :

1. Vérifier le respect des préconditions : afficher la valeurs des paramètres en début de fonction (et peut-être leur types) et / ou écrire du code qui vérifie les préconditions explicitement.
2. Si les paramètres semblent corrects, afficher les valeurs retournées avant chaque instruction `return` et vérifier les résultats à la main.
3. Si la fonction semble correcte, regardez l'appel de la fonction et vérifiez que la valeur de retour est utilisée correctement (ou qu'elle est vraiment utilisée).
4. Afficher certains messages au début et à la fin d'une fonction peut aider à rendre plus visible le flot d'exécution.

*Remarque* : plus tard, vous utiliserez des *débogueurs* qui permettent de "tracer" un programme pas à pas et de consulter les valeurs des variables à chaque étape. Vous pouvez également utiliser Python Tutor pour des petits morceaux de code.

## 5.6. Exercice interactif

*Exercice.* Dans le code suivant,

- Où est le cas de base de la fonction récursive `mystery` et comment y arrive-t-on?
- Que va dessiner la tortue?

```
from uturtle import *

def mystery(t, x, y):
 moveForward(t, x)
 turnRight(t, y)
 if x > 0:
```

1. conditions que la fonction suppose vraies pour les entrées, avant d'exécuter son travail
2. conditions qu'une fonction garantit si les préconditions sont respectées, la validité de son résultat

```
mystery(t, x - 1, y)

bob = umonsTurtle()
mystery(bob, 50, 15)
```

## 5.7. Glossaire

*récurtivité* : le fait pour une fonction de s'appeler elle-même

*cas de base* : branche conditionnelle dans une fonction récursive qui ne fait pas d'appel récursif

*réursion infinie* : fonction qui s'appelle elle-même indéfiniment (sans atteindre jamais un cas de base). Provoque une exception.

*paramètre par défaut* : valeur par défaut que l'on assigne à un paramètre et qui le rend optionnel.

*argument mot-clef* : argument nommé lors d'un appel à une fonction. Cela permet de modifier l'ordre des arguments passés lors de l'appel.

## Itérations et Chaînes de caractères

*Incrémenter et décrémenter • Boucles while • Un str est une séquence immuable de caractères • Boucles for • Invocation de méthodes sur les chaînes • Opérateurs sur les chaînes • Erreurs fréquentes et debug • Exercices interactifs : jouer avec les mots • Pour aller plus loin (à lire par soi-même) • Glossaire*

### 6.1. Incrémenter et décrémenter

#### Assignment multiple

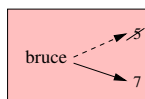
Comme nous l'avons vu, on peut réassigner une valeur à une variable (*assignment multiple*).

```
>>> bruce = 5
>>> print(bruce)
```

5

```
>>> bruce = 7
>>> print(bruce)
```

7



#### Incrémenter et décrémenter

Une des formes les plus courantes de l'assignation multiple : la *mise à jour* d'une variable, où la nouvelle valeur de la variable dépend de son ancienne valeur.

```
x = x + 1
```

*“obtenir la valeur courante de x, ajouter 1, et mettre à jour x avec la nouvelle valeur.”*

Si on met à jour une variable qui n'a pas été créée, cela provoque une erreur (Python



évalue le côté droit de l'assignation *avant* d'assigner la valeur au membre gauche).

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

```
NameError: name 'x' is not defined
```

Avant de mettre à jour une variable, il faut qu'elle ait été *initialisée*, en général avec une simple assignation.

```
>>> x = 0
>>> x = x + 1
```

On dit qu'on *incrémente* une variable quand on met à jour une variable en lui *ajoutant* un nombre (souvent 1).

```
x = x + 1
```

```
step = 2
y = y + step
```

On dit qu'on *décrémente* une variable quand on met à jour une variable en lui *soustrayant* un nombre (souvent 1).

```
n = n - 1
```

## 6.2. Boucles while

La récursivité permet d'automatiser des tâches répétitives. Les *boucles* ou *structures itératives* également.

Répéter des tâches identiques ou similaires sans erreur est quelque chose que les ordinateurs font bien, contrairement aux humains.

Une première forme de structure itérative : boucle `while` ("tant que")

*Syntaxe*

```
while expression booléenne :
 instructions
```

*Comportement* Les instructions (indentées) sont exécutées de manière répétée tant que l'expression booléenne (la condition) retourne vrai.

Voici une manière *récursive* d'afficher un décompte :

```
>>> def countdown(n):
 if n == 0:
 print('Boum')
 else:
 print(n, end = ' ')
 countdown(n - 1)
```

```
>>> countdown(3)
3 2 1 Boum
```

*Remarque* : l'utilisation de `end = ' '` comme dernier argument de la fonction `print` permet de remplacer le retour à la ligne (ici, par un espace).

Voici une version *itérative* de la fonction `countdown` :

```
>>> def countdown(n):
 while n > 0:
```

```

 print(n, end = ' ')
 n = n - 1
 print('Boum')

```

```

>>> countdown(3)
3 2 1 Boum

```

*Remarque* : tout algorithme récursif peut être réécrit sous une forme itérative, et vice versa (cf. Chap. 5 et preuve de Turing).

Plus formellement, voici le flot d'exécution d'une instruction `while` :

1. Evaluer la condition (`True` ou `False`)
2. Si la condition est fausse, sortir de l'instruction `while` et continuer l'exécution à l'instruction suivante
3. Si la condition est vraie, exécuter le *corps* (instructions indentées) de l'instruction `while`, puis retourner à l'étape 1

D'où le nom de *boucle* : dans l'étape 3, on boucle le flot d'exécution vers l'étape 1.

*Exemple* : Version itérative de la méthode de héron (voir chapitre précédent) :

```

def square_root(a):
 x = 0.0
 y = 1.0
 while not almost_equal(x, y):
 x = y
 y = (x + a / x) / 2
 return x

```

### Boucles infinies

Le corps d'une boucle doit mettre à jour les valeurs d'une ou plusieurs variables de telle sorte que la condition soit évaluée à `False` à un moment donné, pour que la boucle se termine.

Dans le cas de `countdown`, on peut prouver facilement que si les préconditions sont respectées ( $n$  est un entier positif fini) alors la boucle se termine.

On parle de preuve de l'*arrêt* d'une boucle ou d'un algorithme contenant des boucles. Bien souvent, ces preuves se font par récurrence.

### Prouver qu'une boucle s'arrête

```

def countdown(n):
 while n > 0: #1
 print(n, end = ' ') #2
 n = n - 1 #3
 print('Boum')

```

*Preuve de l'arrêt de countdown (par récurrence)*. On suppose que  $n$  est un entier  $\geq 0$  (fini). Si  $n = 0$ , la condition (1) est évaluée à faux, et la boucle se termine.

Supposons que la boucle s'arrête pour  $n \leq k$  ( $k \geq 0$ ), et prouvons le pour  $n = k + 1$ . Quand  $n = k + 1$ , l'instruction 2 est exécutée (affiche  $k + 1$ ), puis  $n = k$  lors de l'instruction 3.

La boucle est répétée avec  $n = k$ . Celle-ci s'arrêtera par hypothèse de récurrence. □

Prouver qu’une boucle s’arrête n’est pas toujours facile. Le cas suivant est un exemple (heureusement très rare).

```
def sequence(n):
 while n != 1:
 print(n, end = ' ')
 if n % 2 == 0:
 n = n // 2
 else:
 n = n * 3 + 1
```

- $n$  diminue quand il est pair, mais augmente quand il est impair : une preuve simple par récurrence ne marcherait pas ;
- pour certaines valeurs de  $n$ , il est facile de prouver qu’on atteint 1 (par ex., si  $n$  est une puissance de 2) ;
- mais prouver le cas général (tout entier fini strictement positif) est-il facile ?

**Définition.** Pour un nombre naturel strictement positif  $n$  donné, on peut appliquer l’opération suivante :

- si  $n$  est pair, on le divise par 2 ;
- si  $n$  est impair, on le multiplie par 3 et on ajoute 1.

La *suite de Syracuse* de  $n$  est la suite de nombres naturels obtenue en répétant successivement l’opération ci-dessus.

*Exemple* : à partir de 7, on obtient la suite 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1, ...

**Conjecture** (Conjecture de Syracuse ou de Collatz). *La suite de Syracuse d’un nombre naturel strictement positif quelconque  $n$  atteint toujours 1.*

Cette conjecture constitue un problème ouvert depuis 1952 !

« Les mathématiques ne sont pas encore prêtes pour de tels problèmes » (Erdős)

Une *conjecture* est un énoncé qui semble vrai, mais pour lequel on n’a pas encore pu

- prouver mathématiquement sa validité ;
- trouver de contre-exemple montrant qu’il est faux.

## 6.3. Un str est une séquence immuable de caractères

### Une chaîne de caractères est une séquence

Un `str` est une *séquence* de caractères (collection ordonnée).

On peut accéder à chaque caractère en utilisant l’opérateur `[.]` (crochets droits, “bracket”).

L’opérateur `[x]` — où  $x$  est un entier — retourne un `str` constitué d’un seul caractère : celui se trouvant à l’indice  $x$

*Exemple* :

```
>>> fruit = 'banane'
>>> lettre = fruit[1]
>>> type(lettre)
```

```
<class 'str'>
```

*Question* : que va afficher la ligne suivante ?

```
>>> print(lettre)
```

### Indices d'une chaîne de caractères

```
>>> fruit = 'banane'
>>> lettre = fruit[1]
>>> print(lettre)
```

a

La première lettre du mot banane est “b”, pas “a”. Mais, en informatique, les indices d'une séquence commencent souvent à 0. C'est le cas également en Python.

```
>>> lettre = fruit[0]
>>> print(lettre)
```

b

*Intuition* : indice = nombre de caractères qui séparent le caractère considéré du début de la séquence

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| b | a | n | a | n | e |
| 0 | 1 | 2 | 3 | 4 | 5 |

La *longueur* d'une séquence est son nombre d'éléments. Elle peut être obtenue grâce à la fonction `len` (“length”).

Indices d'une chaîne de longueur  $n$  : entier allant de 0 à  $n - 1$ .

```
>>> mot = 'computer'
>>> len(mot)
8
>>> mot[0]
'c'
>>> mot[7]
'r'
>>> mot[8]
IndexError: string index out of range
>>> mot[1.5]
TypeError: string indices must be integers, not float
```

On peut également accéder aux éléments grâce à un *indice inversé* : le dernier caractère s'obtient avec l'indice  $-1$ , l'avant dernier avec  $-2$ , etc.

Indices inversés d'une chaîne de longueur  $n$  : entier allant de  $-1$  à  $-n$ .

```
>>> mot = 'banane'
>>> mot[-1]
'e'
>>> mot[-6]
'b'
>>> mot[-7]
IndexError: string index out of range
```

|    |    |    |    |    |    |
|----|----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4  | 5  |
| b  | a  | n  | a  | n  | e  |
| -6 | -5 | -4 | -3 | -2 | -1 |

Accéder au dernier élément d'une chaîne est pratique via l'indice  $-1$ , mais pour les autres éléments, l'utilisation est plutôt rare et nous l'éviterons dans ce cours.

```
>>> mot = 'banane'
>>> mot[-1]
'e'
```

### Exercices

*Problème.* Ecrire une fonction qui prend un mot en paramètre et retourne une chaîne qui a remplacé les 'a' par 'o' dans le mot.

*Exemple :* `a2o('haha') → 'hoho'`

Ramplacer les 'a' par 'o' :

```
def a2o(s):
 res = '' # chaîne vide
 i = 0
 while i < len(s):
 if s[i] == 'a':
 res = res + 'o'
 else:
 res = res + s[i]
 i = i + 1
 return res
```

*Problème.* Ecrire une fonction qui prend un mot et une lettre en paramètres et retourne l'indice de la première occurrence de la lettre dans le mot, ou `None` si la lettre n'est pas présente.

*Exemples :* `find('banane', 'a') → 1` `find('banane', 'm') → None`

Trouver l'indice de la première occurrence d'un caractère :

```
def find(word, letter):
 i = 0
 while i < len(word):
 if word[i] == letter:
 return i
 i = i + 1
 return None
```

*Rappel :* `return` provoque une sortie directe de la fonction. C'est également le cas dans une boucle. Donc ici, dès que la lettre est trouvée, on sort de la fonction en retournant l'indice.

La fonction `find` fait donc le contraire de l'opérateur `[.]`, sauf qu'elle ne retourne que le *premier* indice d'une lettre donnée.

```
>>> mot = 'banane'
>>> find(mot, 'a')
1
>>> mot[1]
'a'
>>> mot[3]
'a'
```

*Problème.* Ecrire une fonction qui prend une chaîne en argument et qui retourne une chaîne constituée des caractères de la chaîne dont l'ordre est inversé.

*Exemple* : `invert('algorithmes') → 'emhtirogla'`

Inversion d'une chaîne :

```
def invert(s):
 res = ''
 i = len(s) - 1
 while i >= 0:
 res = res + s[i]
 i = i - 1
 return res
```

*Problème.* Ecrire une fonction qui prend un nombre binaire (entier) en argument (représenté sous la forme d'une chaîne de caractères) et qui retourne ce nombre en base 10 (sous la forme d'un entier). Cette fonction retourne `None` si la chaîne ne représente pas un nombre binaire.

*Exemples* : `bin2dec('10011') → 19` `bin2dec('123') → None`

Conversion binaire → décimal (première version :  $i$  va de  $n - 1$  à 0)

```
def bin2dec(b):
 res = 0
 n = len(b)
 i = n - 1
 puissance = 0
 while i >= 0:
 if b[i] == '1':
 res = res + 2 ** puissance
 elif b[i] != '0':
 return None
 puissance = puissance + 1
 i = i - 1
 return res
```

Conversion binaire → décimal (deuxième version :  $i$  va de 0 à  $n - 1$ )

```
def bin2dec(b):
 res = 0
 i = 0
 n = len(b)
 while i < n:
 if b[n-i-1] == '1':
 res = res + 2 ** i
 elif b[n-i-1] != '0':
 return None
 i = i + 1
 return res
```

### Tranches de chaînes

Un segment d'une chaîne est appelé une *tranche* ("slice"). On peut accéder à une tranche

d'une chaîne avec l'opérateur `[n:m]` où

- `n` est l'indice du premier caractère de la tranche (inclus dans celle-ci)
- `m` est l'indice du dernier caractère de la tranche (*non inclus* dans celle-ci)

```
>>> mot = 'computer'
>>> mot[1:4]
'omp'
>>> mot[0:len(mot)]
'computer'
```

*Intuition* : on peut voir l'opérateur `[n:m]` comme l'intervalle `[n,m[` (fermé – ouvert)

*Remarque* : le nombre de caractères de la tranche est égal à  $m - n$  (quand les indices sont positifs)

Si le premier indice est omis, la tranche commence au début de la chaîne (équivalent à 0)

Si le deuxième indice est omis, la tranche se termine à la fin de la chaîne (équivalent à `len(.)`)

```
>>> mot = 'computer'
>>> mot[:4]
'comp'
>>> mot[4:]
'uter'
>>> mot[:-1]
'compute'
>>> mot[:]
'computer'
```

Si le premier indice est plus grand ou égal au deuxième indice, le résultat est une *chaîne vide*, représentée par deux apostrophes.

```
>>> mot = 'computer'
>>> mot[3:3]
''
```

Une chaîne vide ne contient pas de caractère et a une longueur 0.

A part ça, c'est une chaîne de caractères comme les autres.

## Exercices

*Problème.* Améliorez la fonction `bin2dec` pour gérer également les nombres binaires fractionnaires.

*Exemples* : `bin2dec('11')`  $\rightarrow$  3 `bin2dec('.1')`  $\rightarrow$  0.5 `bin2dec('101.01')`  $\rightarrow$  5.25 `bin2dec('3.1416')`  $\rightarrow$  None

*Hint* : on peut utiliser notre fonction `find` pour chercher la présence d'un point et séparer la chaîne en une tranche entière et une tranche fractionnaire.

L'ancien code de `bin2dec` est placé dans une fonction annexe.

```
def bin2dec_intPart(b):
 res = 0
 i = 0
 n = len(b)
 while i < n:
 if b[n-i-1] == '1':
 res = res + 2 ** i
 elif b[n-i-1] != '0':
 return None
 i = i + 1
```

```
return res
```

*Exemple* : `bin2dec_intPart('101')`  $\rightarrow$  5

On crée une deuxième fonction annexe pour gérer la partie fractionnaire

```
def bin2dec_fracPart(b):
 f = 0.0
 n = len(b)
 i = 0
 while i < n:
 if b[i] == '1':
 f = f + 1.0 / (2 ** (i+1))
 elif b[i] != '0':
 return None
 i = i + 1
 return f
```

*Exemple* : `bin2dec_fracPart('01')`  $\rightarrow$  0.25

On gère le cas général.

```
def bin2dec(b):
 indexDot = find(b, '.')
 if indexDot == None:
 return bin2dec_intPart(b)
 i = bin2dec_intPart(b[:indexDot])
 f = bin2dec_fracPart(b[indexDot + 1:])
 if i != None and f != None:
 return i + f
 else:
 return None
```

*Exemples* : `bin2dec('101')`  $\rightarrow$  5 `bin2dec('.01')`  $\rightarrow$  0.25 `bin2dec('101.01')`  $\rightarrow$  5.25  
`bin2dec('3.101')`  $\rightarrow$  None

## Une chaîne est immuable

On peut être tenté d'utiliser l'opérateur `[.]` dans le membre gauche d'une assignation, pour modifier un de ses caractères.

```
>>> greeting = 'bonjour'
>>> greeting[0] = 'B'
TypeError: 'str' object does not support item assignment
```

Cela provoque une erreur : les chaînes sont *immuables*, c-à-d qu'on ne peut pas modifier une chaîne existante.

*Solution* : créer une nouvelle chaîne et assigner celle-ci à l'ancienne variable.

```
>>> greeting = 'bonjour'
>>> new_greeting = 'B' + greeting[1:]
>>> greeting = new_greeting
>>> print(greeting)
Bonjour
```

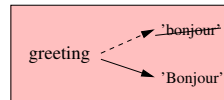
*Remarques* :

- on peut se passer de la variable `new_greeting`
- l'assignation ne modifie pas la chaîne de départ (valeur) mais précise qu'une nouvelle valeur doit être référencée (le membre droit d'une assignation étant évalué avant d'affecter la valeur obtenue)

```
>>> greeting = 'bonjour'
>>> greeting = 'B' + greeting[1:]
```



```
>>> print(greeting)
Bonjour
```



## 6.4. Boucles for

### Objets itérables

Une chaîne de caractère est un objet *itérable*, c'est à dire qu'on peut facilement accéder à chaque élément de la séquence, un par un.

Le moyen le plus simple d'itérer est d'utiliser une boucle `for` : chaque *itération* de la boucle accèdera à un élément de la séquence via une variable qui est mise à jour à chaque itération.

```
>>> for lettre in 'bonjour':
 print(lettre, end = ', ')
```

```
b, o, n, j, o, u, r,
```

### Boucles for

#### Syntaxe

```
for variable in itérable :
 instructions
```

**Comportement** Le corps de la boucle (instructions indentées) est exécuté de manière répétée pour chaque élément de l'objet itérable : à chaque itération, un élément est affecté à la variable.

#### Exemple

```
>>> s = 'hello'
>>> res = ''
>>> for c in s:
 res = res + c + c

>>> print(res)
hheelllloo
```

Il existe beaucoup de types d'objets itérables en Python.

Par exemple : les tuples (ch. 3 et 11), les listes (ch. 7), les dictionnaires (ch. 10) et les fichiers (ch. 12).

```
>>> t = (2, 3)
>>> type(t)
<class 'tuple'>
>>> res = 0
>>> for x in t:
 res = res + x

>>> print(res)
5
```

**Objets** `range`

La fonction `range(n, m)` retourne un objet `range` qui, quand il est itéré, permet d'accéder aux entiers de l'intervalle `[n, m[`.

Les objets `range` sont également itérables.

```
>>> type(range(1, 4))
<class 'range'>
>>> for i in range(1, 4):
 print(i, end = ' ')
```

```
1 2 3
```

*Remarque :* en Python 2, la fonction `range` retourne une liste au lieu d'un objet `range`.

Si un seul paramètre `m` est donné à la fonction `range`, l'intervalle est `[0, m[`.

```
>>> for i in range(5):
 print(i, end = ' ')

0 1 2 3 4
>>> s = 0
>>> for i in range(6):
 s = s + i

>>> print(s)
15
>>> for i in range(3):
 print('he', end = '')

hehehe
```

**Exercice**

*Problème.* Ecrire une fonction qui compte le nombre d'occurrences d'une lettre donnée dans une chaîne.

*Exemple :* `nbr_occurences('banane', 'n') → 2`

```
def nbr_occurences(word, letter):
 cnt = 0
 for char in word:
 if char == letter:
 cnt = cnt + 1
 return cnt
```

## 6.5. Invocation de méthodes sur les chaînes

**Objets**

Dans ce cours, on a déjà évoqué la notion d'*objet* (par ex., objets "module" et objets "fonctions").

```
>>> import math
>>> print(math)
<module 'math' from '/Library/python2.6/lib-dynload/math.so'>
>>> print(nbr_occurences)
<function nbr_occurences at 0xead7b0>
```

C'est limitatif mais — pour le moment — considérons un objet comme un valeur (assignée à une variable) possédant des *attributs*.

*Exemple* : Un objet de type “voiture” pourrait avoir les attributs :

marque, cylindrée, prix, couleur, etc.

La notation point permet d'accéder aux attributs d'un objet

*Exemple* : variables et fonctions d'un objet module.

```
>>> math.sin(math.pi / 2.0)
1.0
```

La fonction `dir` permet d'afficher la liste des attributs d'un objet.

```
>>> dir(nbr_occurences)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
 '__defaults__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__get__', '__getattr__', '__globals__',
 '__gt__', '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__',
 '__module__', '__name__', '__ne__', '__new__', '__qualname__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__']
>>> print(nbr_occurences.__name__)
nbr_occurences
```

En réalité, *en Python, tout est objet*. Cela implique que tout peut être assigné à une variable ou passé comme argument à une fonction, même une fonction ou un module.

Les entiers et les chaînes de caractères sont donc aussi des objets.

```
>>> s = 'bonjour'
>>> dir(s)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
...
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

Les attributs sont liés au type des objets, c'est à dire qu'ils sont

- *identiques pour tous les objets d'un même type*;
- *spécifiques au type de l'objet, donc deux objets de deux types différents n'ont pas la même liste d'attributs*.

*En séance* : illustration des sorties des appels à `dir` suivants :

```
>>> s = 'bonjour'
>>> dir(s)
...
>>> dir(str)
...
>>> dir('')
...
>>> dir(int)
...
>>> dir(12)
...
```

## Méthodes

Les attributs d'un objet `str` sont nombreux et de types divers, mais la plupart d'entre eux

sont des *méthodes*.

```
>>> s = 'bonjour'
>>> type(s.__doc__)
<class 'str'>
>>> type(s.upper)
<class 'builtin_function_or_method'>
>>> type('hello'.find)
<class 'builtin_function_or_method'>
```

Une *méthode* est similaire à une fonction — elle prend des arguments et retourne une valeur — mais elle s'applique *sur* un objet, et la syntaxe est donc légèrement différente.

### Invocations de méthodes

Pour obtenir une (nouvelle) chaîne en majuscules à partir d'une chaîne, on peut :

- écrire, puis utiliser une fonction

```
>>> def set_upper(s):
 ...
```

```
>>> res = set_upper('bonjour')
>>> print(res)
'BONJOUR'
```

- invoquer la méthode `upper` qui est disponible pour tous les objets de type `str`

```
>>> s = 'bonjour'
>>> res = s.upper()
>>> print(res)
'BONJOUR'
>>> print('hello'.upper())
'HELLO'
```

On parle d'*appel* à une fonction. Une fonction a besoin de toutes ses entrées en paramètres (ici, une chaîne).

```
>>> res = set_upper('bonjour')
```

On dit qu'on *invoque* une méthode *sur* un objet.

```
>>> res = 'bonjour'.upper()
```

La notation point permet de préciser sur quel objet la méthode est invoquée. Il n'est donc pas nécessaire de passer cet objet en paramètre.

Excepté cette différence de syntaxe, les méthodes et les fonctions s'utilisent de façon similaire (parenthèses obligatoires, arguments, valeur de retour, etc.).

### Exemples de méthodes sur les chaînes

```
>>> print(str.islower.__doc__)
S.islower() -> bool
```

Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.

```
>>> 'hello'.islower()
True
>>> 'Hello'.islower()
False
>>> '1'.isdigit()
True
>>> 'a'.isdigit()
False
```

Notre fonction `find` devient obsolète grâce à la méthode `find` prédéfinie :

```
>>> find('banane','n')
2
>>> 'banane'.find('n')
2
```

Cette méthode est plus évoluée que notre fonction `find` :

```
>>> print(str.find.__doc__)
S.find(sub [,start [,end]]) -> int
```

Return the lowest index in S where substring sub is found, such that sub is contained within s[start:end]. Optional arguments start and end are interpreted as in slice notation.

```
>>> mot = 'banane'
>>> mot.find('an')
1
>>> mot.find('an', 2)
3
```

*Exercice* : lisez la documentation de la méthode `count` pour écrire une instruction qui permette de compter le nombre de “n” du mot “banane”. Qu’est-ce qui est obligatoire et qu’est-ce qui est optionnel ? Comment le voit-on dans l’en-tête de la méthode ?

```
>>> print(str.count.__doc__)
S.count(sub[, start[, end]]) -> int
```

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

## 6.6. Opérateurs sur les chaînes

### Opérateur `in`

Les opérateurs `+` et `*` peuvent être appliqués sur des `str` (voir ch. 2). Il existe d’autres opérateurs pour les chaînes.

“`in`” n’est pas utilisé que dans une boucle `for`. C’est aussi un opérateur booléen qui retourne vrai si un élément est inclus dans une séquence. Dans le cas des chaînes, il retourne vrai si une sous-chaîne est incluse dans une chaîne.

```
>>> 'a' in 'banane'
True
>>> 'pepin' in 'banane'
False
```

Par exemple, la fonction suivante affiche toutes les lettres d’un mot qui apparaissent aussi dans un deuxième mot :

```
>>> def in_both(word1, word2):
 for letter in word1:
 if letter in word2:
 print(letter, end = ' ')

>>> in_both('pommes','oranges')
o e s
```

*Attention* : aux deux façons d’utiliser `in`

- le premier, avec `for`, pour itérer sur chaque lettre
- le deuxième signifie “est-ce une sous-chaîne?”

## Comparaison de chaînes

L'opérateur de comparaison `==` fonctionne sur les chaînes :

```
if answer == 'Thank you':
 print('You are welcome')
```

Les autres opérateurs de comparaison peuvent être également utilisés sur des chaînes : dans ce cas, “plus petit” ou “plus grand” signifie “précède” ou “suit” dans l’ordre alphabétique.

```
>>> 'avoir' < 'etre'
True
>>> 'elephant' > 'souris'
False
>>> 'avoir' < 'Etre'
False
```

*Remarque :* en Python, toutes les lettres majuscules viennent avant les lettres minuscules. La méthode `lower` peut être utile si c’est un problème.

*Problème.* Ecrire une fonction qui permet de trier trois mots, quelle que soit la casse des mots.

On commence par écrire une fonction qui permet de trier deux mots et qui gère le problème de la casse.

```
>>> def trie2(mot1, mot2):
 first = mot1
 second = mot2
 if mot1.lower() > mot2.lower():
 first = mot2
 second = mot1
 return (first, second)

>>> trie2('Zebre', 'elephant')
('elephant', 'Zebre')
```

On peut maintenant trier trois mots sans se soucier de la casse : on les trie deux à deux.

```
>>> def trie3(mot1, mot2, mot3):
 (mot1, mot2) = trie2(mot1, mot2)
 # le plus grand de mot1 et mot2 est devenu mot2

 (mot2, mot3) = trie2(mot2, mot3)
 # mot3 est le plus grand des 3 mots

 (mot1, mot2) = trie2(mot1, mot2)
 # il se peut que (le nouveau) mot2 soit plus petit que mot1

 return (mot1, mot2, mot3)

>>> trie3('a', 'c', 'b')
('a', 'b', 'c')
>>> trie3('c', 'b', 'a')
('a', 'b', 'c')
```

## 6.7. Erreurs fréquentes et debug

### Erreurs d’indices

Quand on utilise les indices pour traverser une séquence, les erreurs d’indices sont fréquentes (indice de fin ou de début).

Voici une fonction qui est supposée comparer deux mots et retourner `True` si l'un des mots est l'inverse de l'autre, mais elle contient deux erreurs :

```
def is_reverse(word1, word2):
 if len(word1) != len(word2):
 return False
 i = 0
 j = len(word2)
 while j > 0:
 if word1[i] != word2[j]:
 return False
 i = i + 1
 j = j - 1
 return True
```

- le premier `if` vérifie que les 2 mots ont la même longueur, si ce n'est pas le cas, ils ne peuvent être inverses l'un de l'autre
- `i` et `j` sont des indices : `i` traverse `word1` de g. à dr. et `j` traverse `word2` de dr. à g.
- le deuxième `if` vérifie que les lettres “miroir” sont équivalentes, si ce n'est pas le cas, les mots ne peuvent être inverses l'un de l'autre
- si on traverse toute la boucle et que toutes les lettres correspondent, on peut retourner `True`

Si on teste la fonction avec les mots “pots” et “stop”, on s'attend à obtenir `True`, mais on obtient une erreur d'indice :

```
>>> is_reverse('pots', 'stop')
...
File "<pyshell#211>", line 7, in is_reverse
 if word1[i] != word2[j]:
IndexError: string index out of range
```

Pour déboguer les erreurs d'indices, un réflexe utile est d'afficher les indices.

```
...
 while j > 0:
 print('i:', i, 'j:', j)
 ...
```

On observe que lors de la première itération, l'erreur est déjà provoquée. Il s'agit de l'indice de `j` qui est plus grand que `len('stop')-1` :

```
>>> is_reverse('pots', 'stop')
i: 0 j: 4
...
 if word1[i] != word2[j]:
IndexError: string index out of range
```

*Correction :*

```
def is_reverse(word1, word2):
 ...
 j = len(word2) - 1
 ...
```

On obtient :

```
>>> is_reverse('pots', 'stop')
i: 0 j: 3
i: 1 j: 2
i: 2 j: 1
True
```

La réponse est bonne mais la boucle itère 3 fois : est-ce normal ? La fonction est-elle correcte ?

*Exercice* : utilisez Python Tutor pour corriger complètement cette fonction.

## 6.8. Exercices interactifs : jouer avec les mots

### Exercice any\_lowercase

*Exercice.* Un étudiant écrit 5 versions d’une fonction `any_lowercase(s)` qui a pour but de tester si une chaîne `s` contient au moins une lettre minuscule. Quelles sont les versions qui sont correctes ? Que font réellement les mauvaises versions ?

```
def any_lowercase1(s):
 for c in s:
 if c.islower():
 return True
 else:
 return False

def any_lowercase2(s):
 for c in s:
 if 'c'.islower():
 return True
 else:
 return False

def any_lowercase3(s):
 for c in s:
 if c.islower():
 return True
 return False

def any_lowercase4(s):
 for c in s:
 flag = c.islower()
 return flag

def any_lowercase5(s):
 flag = False
 for c in s:
 flag = flag or c.islower()
 return flag
```

Le fichier *words.txt*<sup>1</sup> (disponible sur e-learning) 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)
<_io.TextIOWrapper name='words.txt' mode='r' encoding='UTF-8'>
```

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

1. Fichier disponible publiquement, fourni par Grady Ward pour le projet “Moby”



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”<sup>2</sup> : 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 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.

Un fichier est un objet itérable. Une boucle `for` permet de 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
```

Depuis Python 3, utiliser `with open...` permet d’ouvrir un fichier mais également de gérer automatiquement certaines choses (comme la fermeture du fichier à la fin du `with` même si une exception se produit).

```
cnt = 0
with open("words.txt") as file:
 for line in file:
 cnt += 1
print(cnt)
```

## Exercices

*Problème.* Ecrivez 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).

*Problème.* En 1939, Ernest Wright a publié une nouvelle de 50000 mots appelée *Gadsby* qui

---

2. D’autres caractères blancs : un espace ou une tabulation `\t`

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<sup>3</sup>.

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`.

*Solution* : les mots de plus de 20 caractères :

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

*Solution* : les mots sans “e” :

```
with open('words.txt') as fichier:
 total = 0
 cnt = 0
 for line in fichier:
 total = total + 1
 word = line.strip()
 if not 'e' in word:
 cnt = cnt + 1

percent = cnt / total * 100.0
print('Pourcentage de mots sans e: %.2f' % percent)
```

*Problème.* Donnez-moi 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. Pourrez-vous me dire, pour le prochain cours, quel est le mot auquel je pense ?

## 6.9. Pour aller plus loin (à lire par soi-même)

### Assignment multiple et test d’égalité

Avec les assignments multiples, il faut faire d’autant plus attention à ne pas confondre l’égalité au sens mathématique et l’assignment.

|            | Relation d’égalité                                                                                     | Assignment                                                                                                                                                  |
|------------|--------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Symétrique | Oui : Si $a = 7$ , alors, $7 = a$                                                                      | Non : $a = 7$ légal, $7 = a$ pas                                                                                                                            |
| Booléen    | Oui : égalité est soit vraie, soit fausse.<br>En math, si $a = b$ , alors $a$ sera toujours égal à $b$ | Non. De plus, peut rendre deux variables égales, mais cela peut changer<br>$a = 5$<br>$b = a$ # $a$ et $b$ égaux<br>$a = 3$ # $a$ et $b$ ne sont plus égaux |

### Un peu plus de détails sur `range`

3. 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”.

La fonction `range` retourne un objet `range` permettant d’itérer sur une suite d’entiers : `range(start, stop, step)`

- `start` (optionnel) : premier entier de la suite (par défaut 0)
- `stop` (obligatoire) : dernier entier de la suite *non-compris*
- `step` (optionnel) : entier qui représente l’incrément ou le décrétement entre deux valeurs de la suite (par défaut 1)

```
>>> for i in range(1, 5):
 print(i, end = ' ')

1 2 3 4
>>> for i in range(5):
 print(i, end = ' ')

0 1 2 3 4
>>> for i in range(2, 8, 2):
 print(i, end = ' ')

2 4 6
>>> for i in range(8, 0, -1):
 print(i, end = ' ')

8 7 6 5 4 3 2 1
>>> for i in range(stop = 9, step = 3):
 print(i, end = ' ')
```

`TypeError: range() does not take keyword arguments`

La fonction `range` ne prend pas d’arguments mots-clefs : il faut donc spécifier `start` si on veut définir `step`.

### Exemple d’utilisation de `range`

*Problème.* Ecrire une fonction qui calcule la somme des nombres pairs présents dans l’intervalle  $[n, m]$  ( $n \leq m$ ).

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

>>> sum_even(2, 4)
6
>>> sum_even(1, 5)
6
```

### Debug par bisection

Quand la taille d’un programme augmente, il en va de même pour le travail de debug. Une manière de gagner du temps pour trouver un bug, est la méthode dite de “bisection”. Par exemple,

- s’il y a 100 lignes de code, et que vous le testez ligne après ligne, cela fera 100 étapes.
- essayez plutôt de couper le problème en deux. Inspectez au milieu du programme une valeur que vous pouvez tester (ajoutez un `print` ou quelque chose qui a un

effet vérifiable).

- si le test au milieu est incorrect, le problème devrait se trouver dans la première partie du programme. S'il est correct, il devrait se trouver dans la seconde partie.
- répétez la procédure jusqu'à la localisation précise du problème.
- en pratique, il n'est pas toujours facile (ou possible) de tester le "milieu" du programme. Dans ce cas, choisissez des endroits où ajouter un test est facile.

### Données pour les tests unités

- des fichiers comme `words.txt` sont utiles pour créer de nombreux tests unités automatiquement.
- s'assurer de tester tous les "cas spéciaux" : par ex., une méthode qui vérifie si un caractère est présent dans une chaîne devrait tester les cas où : le caractère est présent en début, en fin et au milieu ; tester la chaîne vide ; tester le cas où le caractère n'est pas présent.
- tester aide à trouver des bugs, mais il n'est pas toujours facile de générer de bons ensembles de cas testés, et même si c'est le cas, vous n'avez jamais l'assurance que le programme est correct.

Dijkstra (informaticien célèbre) :

*Program testing can be used to show the presence of bugs, but never to show their absence !*

## 6.10. Glossaire

*incrémenter* : mettre à jour une variable en augmentant sa valeur (souvent de 1).

*décrémenter* : mettre à jour une variable en diminuant sa valeur (souvent de -1).

*itération* : (ou boucle) exécution répétée d'un ensemble d'instructions.

*boucle infinie* : une boucle dont la condition de fin n'est jamais satisfaite.

*objet* : quelque chose qu'une variable peut référer. Pour le moment, un objet est une "valeur" qui possède des attributs et des méthodes.

*séquence* : ensemble ordonné, c-à-d un ensemble de valeurs où chaque valeur est identifiée par un index entier.

*élément* : (item) une des valeurs d'une séquence.

*tranche* : (slice) partie d'une chaîne spécifiée par un intervalle d'indices.

*chaîne vide* : chaîne sans caractère et de longueur 0, représentée par deux apostrophes.

*immuable* : propriété d'une séquence dont les éléments ne peuvent être assignés.

*méthode* : fonction qui est associée à un objet et qui est invoquée en utilisant la notation point.

---

# Listes

*Une liste est une séquence d'éléments • Listes et opérateurs • Méthodes sur les listes • Opérations fréquentes sur les listes • Objets et valeurs • Arguments de la ligne de commande (parenthèse) • Erreurs fréquentes et debug • Exercices interactifs • Pour aller plus loin (à lire par soi-même) • Glossaire*

---

## Illustration des listes

*Illustration en auditoire* : script permettant d'essayer de résoudre l'énigme de la traversée (deux niveaux) et dont le code sera dévoilé plus tard.

## 7.1. Une liste est une séquence d'éléments

Comme une chaîne de caractères, une *liste* est une séquence de valeurs, mais :

- dans une chaîne les valeurs sont des caractères, dans une liste elles peuvent être de n'importe quel type
- une chaîne est une séquence immuable, une liste est une séquence que l'on peut modifier
- une liste est définie en spécifiant ses éléments, séparés par des virgules, entre crochets (par ex. : `[1, 2, 3]`)

## Création de listes

*Remarques :*

- une liste peut être vide
- une liste peut contenir des listes, des tuples, ou n'importe quelle autre type de valeur
- une liste peut contenir des valeurs de plusieurs types différents
- la fonction `len` peut être appliquée sur une liste

```
>>> empty = []
>>> type(empty)
<class 'list'>
>>> data = ['text', 2.0, 1, [2, 3]]
>>> print(data)
['text', 2.0, 1, [2, 3]]
>>> len(data)
4
```

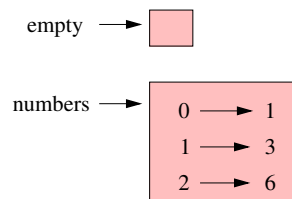
### Accéder et modifier les éléments d'une liste.

On peut accéder aux éléments d'une liste grâce à l'opérateur `[.]`. Les indices fonctionnent de la même manière que pour les chaînes de caractères.

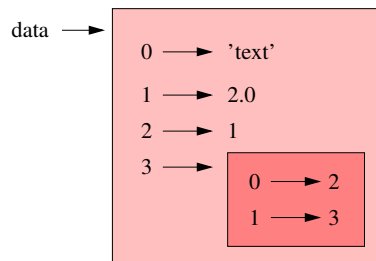
```
>>> data = ['text', 2.0, 1, [2, 3]]
>>> print(data[0])
text
>>> print(data[-1])
[2, 3]
```

On peut représenter les listes par un diagramme d'état qui illustre les relations entre les indices et les éléments.

```
>>> empty = []
>>> numbers = [1, 3, 6]
```



```
>>> data = ['text', 2.0, 1, [2, 3]]
```

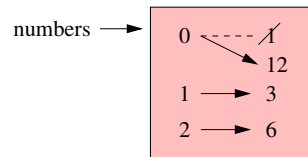


*Démonstration : représentation de data via Python Tutor*

Contrairement aux chaînes qui sont immuables, on peut modifier les éléments d'une liste.

```
>>> mot = 'bonjour'
>>> numbers = [1, 3, 6]
>>> mot[0] = 'B'

Traceback (most recent call last):
 File "<pyshell#36>", line 1, in <module>
 mot[0] = 'B'
TypeError: 'str' object does not support item assignment
>>> numbers[0] = 12
>>> print(numbers)
[12, 3, 6]
```



## Boucles for

Une liste est un objet itérable : elle peut donc être utilisée dans une boucle `for`.

```
data = ['text', 2.0, 1, [2, 3]]
```

```
for item in data:
 print(item, type(item))
```

```
text <class 'str'>
2.0 <class 'float'>
1 <class 'int'>
[2, 3] <class 'list'>
```

## Convertir un objet en liste

La fonction `list` permet de convertir un objet en liste, si c'est possible. Cela fonctionne notamment avec les objets de type `range` et de type `str`.

```
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(2, 6))
[2, 3, 4, 5]
>>> s = 'hello'
>>> list(s)
['h', 'e', 'l', 'l', 'o']
```

Liste de listes  $\simeq$  tableau multidimensionnel.

*Exemple : la matrice*

$$A = \begin{pmatrix} 1 & 5 & 3 \\ 4 & 2 & 2 \\ 3 & 7 & 9 \end{pmatrix}$$

peut être représentée par

```
A = [[1, 5, 3], [4, 2, 2], [3, 7, 9]]
```

L'accès à un élément de la *i*ème ligne, *j*ème colonne se fait via

```
A[i-1][j-1]
```

Par exemple l'élément au centre de la matrice ci-dessus est accédé via

```
A[1][1]
```

## Exercices

*Exercice.* Ecrire une fonction qui retourne la plus grande valeur d'une liste contenant des entiers.

*Précondition :* on suppose que la liste n'est pas vide.

*Exercice.* Modifiez le code de l'exercice précédent pour gérer le cas d'une liste vide. Dans ce cas, il faut retourner `None`.

*Cas avec précondition spécifiant que la liste n'est pas vide :*

```
def max_liste(t):
 max = t[0]
 for x in t:
 if x > max:
 max = x
 return max
```

*Cas avec liste vide gérée :*

```
def max_liste(t):
 if len(t) == 0:
 return None
 else:
 max = t[0]
 for x in t:
 if x > max:
 max = x
 return max
```

*Variante qui utilise les indices :*

```
def max_liste(t):
 if len(t) == 0:
 return None
 else:
 max = t[0]
 for i in range(1, len(t)):
 if t[i] > max:
 max = t[i]
 return max
```

Grâce aux indices, on peut éviter de tester le premier élément.

*Exercice.* Ecrire une fonction qui retourne la plus grande valeur d'une matrice  $n \times m$  contenant des entiers.

*Précondition :* on suppose que  $n$  et  $m$  sont  $\geq 1$  et que la matrice donnée en entrée est correctement codée (c'est à dire une liste de  $n$  listes de  $m$  entiers).

*Variante qui utilise les indices :*

```
def max_matrix(A):
 n = len(A)
 m = len(A[0])
 max = A[0][0]

 for i in range(n):
 for j in range(m):
 if A[i][j] > max:
 max = A[i][j]
 return max
```

```
A = [[1, 5, 3], [4, 2, 2], [3, 7, 9], [6, 8, 9]]
print(max_matrix(A))
```

*Exercice.* Ecrire une fonction qui affiche correctement une matrice  $n \times m$  contenant des entiers dans la console.

*Précondition :* on suppose que  $n$  et  $m$  sont  $\geq 1$  et que la matrice donnée en entrée est correctement codée (c'est à dire une liste de  $n$  listes de  $m$  entiers).

```
def print_matrix(A):
 n = len(A)
 m = len(A[0])

 for i in range(n):
```



```

for j in range(m):
 print(A[i][j], end = ' ')
print('')

```

```

A = [[1, 5, 3], [4, 2, 2], [3, 7, 9], [6, 8, 9]]
print_matrix(A)

```

## 7.2. Listes et opérateurs

Comme pour les chaînes, l’opérateur `+` concatène deux listes, l’opérateur `*` répète une liste un certain nombre de fois.

```

>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
>>> [0] * 4
[0, 0, 0, 0]
>>> a * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

```

### Opérateur `in`

Comme pour toutes les séquences, l’opérateur `in` est applicable sur une liste.

```

>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False

```

### Tranches de listes

L’opérateur `[ : ]` fonctionne aussi sur les listes.

```

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

```

Il peut même être utilisé dans le membre gauche d’une assignation.

```

>>> t[1:3] = [34, 26]
>>> print(t)
[0, 34, 26, 3, 4, 5, 6, 7, 8, 9]

```

### Copie d’une liste

Comme les listes sont mutables, il est parfois utile de faire une copie avant de les modifier. On peut créer une copie avec une “tranche complète”.

```
>>> copy = t[:]
>>> t[4:6] = [0] * 2
>>> print(t, copy)
[0, 34, 26, 3, 0, 0, 6, 7, 8, 9] [0, 34, 26, 3, 4, 5, 6, 7, 8, 9]
```

*Remarque* : comme on va le voir à la fin de ce chapitre, cette méthode ne fonctionne correctement que si la liste ne contient que des objets immuables.

## 7.3. Méthodes sur les listes

Les objets de type `list` possèdent une série de méthodes utiles.

```
>>> dir(list)
[...]
'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

La méthode `append` ajoute un nouvel élément à la fin de la liste.

```
>>> help(list.append)
Help on method_descriptor:

append(...)
 L.append(object) -> None -- append object to end
```

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

La méthode `pop` *supprime et retourne* le dernier élément de la liste.

```
>>> help(list.pop)
Help on method_descriptor:

pop(...)
 L.pop([index]) -> item -- remove and return item at index (default last).
 Raises IndexError if list is empty or index is out of range.
```

```
>>> t = ['a', 'b', 'c', 'd']
>>> deleted = t.pop()
>>> print(deleted)
d
>>> print(t)
['a', 'b', 'c']
```

Si l'argument optionnel est défini, supprime et retourne l'élément situé à l'indice donné.

```
>>> t.pop(1)
'b'
>>> print(t)
['a', 'c']
```

La méthode `sort` trie les éléments d'une liste. Ses arguments sont optionnels : par exemple, `reverse` permet de trier par ordre décroissant.

```
>>> help(list.sort)
Help on method_descriptor:

sort(...)
 L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*

>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

```
['a', 'b', 'c', 'd', 'e']
>>> t.sort(reverse=True)
>>> print(t)
['e', 'd', 'c', 'b', 'a']
```

D'autres méthodes utiles sont présentées dans la section "Pour aller plus loin" à lire par vous même, à la fin de ce chapitre.

## Exercice

*Exercice.* Ecrire un script qui demande à l'utilisateur le nom d'un fichier de texte (.txt), affiche les mots contenus dans le fichier par ordre alphabétique (en minuscules). 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<sup>1</sup>). On peut spécifier d'autres séparateurs que des blancs avec le paramètre optionnel `sep`.

```
>>> help(str.split)
Help on method_descriptor:

split(...)
 S.split(sep=None, maxsplit=-1) -> list of strings

 Return a list of the words in S, using sep as the
 delimiter string. If maxsplit is given, at most maxsplit
 splits are done. If sep is not specified or is None, any
 whitespace string is a separator and empty strings are
 removed from the result.

>>> 'Un deux trois'.split()
['Un', 'deux', 'trois']
>>> '000-123456-47'.split('-')
['000', '123456', '47']
>>> '000-123456-47'.split('-', 1)
['000', '123456-47']
```

On définit une fonction `clean` qui retourne la chaîne passée en argument en minuscules, et en lui ayant retiré tous les caractères qui ne sont pas des lettres ou des chiffres.

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

>>> clean('He#23.?6Xth')
he236xth

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

---

1. Espaces, retours à la ligne, tabulations, etc.

```
wordsList.sort()
print(wordsList)
```

## 7.4. Opérations fréquentes sur les listes

### Réduction

Pour additionner tous les nombres d’une liste, on peut utiliser une boucle :

```
def add_all(t):
 total = 0
 for x in t:
 total += x
 return total

>>> t = list(range(1,11))
>>> add_all(t)
55
```

*Remarques :*

- `total += x` est équivalent à `total = total + x`
- une variable comme `total` qui accumule progressivement la somme des éléments est appelée un *accumulateur*

Additionner les éléments d’une liste est une opération fréquente. Python offre une fonction qui permet également de le faire.

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

Une opération qui combine une séquence d’éléments en une seule valeur est appelée une *réduction*.

Réduction : séquence  $\rightarrow$  valeur.

### Mapping

Autre opération fréquente : traverser une séquence pour en construire une nouvelle. Par exemple, la fonction suivante prend une liste de chaînes en argument et retourne une liste qui “capitalise” les mots de la liste de départ<sup>2</sup>.

```
def capitalize_all(t):
 res = []
 for s in t:
 res.append(s.capitalize())
 return res

>>> liste = ['hello', 'bonjour', 'dag']
>>> capitalize_all(liste)
['Hello', 'Bonjour', 'Dag']

def capitalize_all(t):
 res = []
 for s in t:
 res.append(s.capitalize())
 return res
```

---

2. On utilise la méthode `capitalize` sur chaque chaîne : elle met la première lettre en majuscule.

- `res` est initialisé avec une liste vide ; à chaque itération de la boucle, on ajoute le nouvel élément : `res` est une sorte d'accumulateur.
- Une opération qui applique une fonction (ici, `capitalize`) sur chaque élément d'une séquence est appelée un *mapping*.

Mapping : séquence  $\rightarrow$  séquence.

## Filtre

Autre opération fréquente : filtrer une séquence. Par exemple, la fonction suivante ne retourne que les chaînes qui sont en majuscules.

```
def only_upper(t):
 res = []
 for s in t:
 if s.isupper():
 res.append(s)
 return res

>>> liste = ['Hello', 'bonjour', 'DAG']
>>> only_upper(liste)
['DAG']
```

- une opération qui sélectionne certains éléments pour retourner une sous-séquence est appelée un *filtre*.

Filtre : séquence  $\rightarrow$  sous-séquence.

## Exercice

*Exercice.* Ecrire une fonction qui prend une liste de nombres entiers en paramètres et qui retourne la somme cumulative, c-à-d une nouvelle liste telle que l'élément d'indice  $i$  soit la somme des  $i + 1$  premiers éléments.

Par exemple, la somme cumulative de  $[1, 2, 3]$  est  $[1, 3, 6]$ .

```
def cumul_sum(t):
 res = []
 s = 0
 for x in t:
 s += x
 res.append(s)
 return res

>>> cumul_sum([1, 2, 3])
[1, 3, 6]
>>> cumul_sum(list(range(10)))
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

## Fonctions Python pour les opérations fréquentes

La plupart des opérations sur des séquences peuvent être exprimées comme une combinaison de mappings, de filtres et de réductions.

Python possède des fonctions pour réaliser directement deux de ces opérations : `map`, `filter`. Une réduction est en général réalisée via l'utilisation d'un accumulateur et d'une boucle `for`.

**Fonction** map

```
map(function, *iterables) -> map
```

Applique une action (fonction ou méthode) sur chaque élément d'un objet itérable et retourne un objet map qui est itérable ou peut être converti en liste.

```
>>> import math
>>> list(map(math.sqrt, [2, 4, 6, 100]))
[1.4142135623730951, 2.0, 2.449489742783178, 10.0]
>>> list(map(str.upper, 'hello'))
['H', 'E', 'L', 'L', 'O']
>>> def double(lettre):
 return lettre * 2

>>> for l in map(double, 'Hello'):
 print(l, end = ' ')
```

```
HH, ee, ll, ll, oo,
```

Dans ce code, on a défini une fonction double pour pouvoir utiliser map :

```
>>> def double(lettre):
 return lettre * 2

>>> print(list(map(double, 'Hello')))
['HH', 'ee', 'll', 'll', 'oo']
```

Mais cette fonction est une “mini” fonction, qui pourrait être définie juste là où on a en besoin. En Python, on peut faire cela grâce aux *fonctions lambda*. Le code ci-dessus peut être réécrit en une seule ligne et sans définir double.

```
>>> print(list(map(lambda x: x * 2, 'Hello')))
['HH', 'ee', 'll', 'll', 'oo']
```

**Fonction** filter

```
filter(function or None, iterable) -> filter object
```

Sélectionne les éléments qui satisfont un certain critère (fonction ou méthode booléenne) et les retournés sous forme d'un objet filter qui est itérable ou peut être converti en liste.

```
>>> list(filter(str.isalpha, 'r2d2'))
['r', 'd']
>>> def is_even(x):
 return x % 2 == 0

>>> list(filter(is_even, [1, 2, 4, 5, 7, 10, 11]))
[2, 4, 10]
```

*Remarque* : si la fonction est “None”, retourne les éléments qui sont True.

*Exercice*. Ecrire une fonction qui, à partir d'une liste d'entiers en entrée, retourne une liste contenant les carrés des nombres de cette liste, mais uniquement si ceux-ci se terminent par le chiffre 6.

*Challenge* : le corps de votre fonction peut être écrit en une seule ligne !

*Version longue*

```
def square(x):
 return x ** 2

def six_ended(x):
 return x % 10 == 6
```

```
def six_ended_squares(t):
 squares = list(map(square, t))
 res = list(filter(six_ended, squares))
 return res

print(six_ended_squares(list(range(100))))
```

### Version courte

```
def six_ended_squares(t):
 return list(filter(lambda x: x % 10 == 6, list(map(lambda x: x ** 2, t))))

print(six_ended_squares(list(range(100))))
```

## 7.5. Objets et valeurs

Si on exécute les assignations suivantes, on sait que `a` et `b` réfèrent à une chaîne, mais on ne sait pas si ces variables réfèrent la *même* chaîne.

```
>>> a = 'banana'
>>> b = 'banana'
```

Il y a deux possibilités :



Dans un cas, `a` et `b` réfèrent deux différents objets qui ont la même valeur. Dans l'autre, elles réfèrent au même objet.

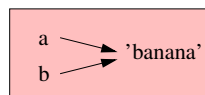
```
>>> a = 'banana'
>>> b = 'banana'
```

L'opérateur `is` permet de tester si deux variables réfèrent le même objet.

Pensez-vous que `a is b` ?

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

Donc, dans ce cas, Python crée un seul objet "chaîne" et les deux variables réfèrent à cet objet.



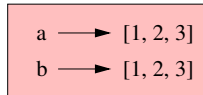
Qu'en est-il des listes ?

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
```

Pensez-vous que `a is b` ?

Dans ce cas, Python crée deux objets différents.

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```



Remarques :

- les deux listes sont *équivalentes*, car elles ont les mêmes valeurs, mais elles ne sont pas *identiques*, car elles ne sont pas le même objet ;
- si deux objets sont identiques, ils sont aussi équivalents, mais le contraire n'est pas toujours vrai.

Pourquoi deux comportements différents en créant deux chaînes ou deux listes ?

```

>>> 'banana' is 'banana'
True
>>> [1, 2, 3] is [1, 2, 3]
False

```

Car les chaînes sont immuables et les listes mutables.

- Si on modifie par après une des deux listes, l'autre n'est pas modifiée : elles ont "pour le moment" la même valeur, mais elles peuvent évoluer de manière indépendante, puisqu'elles sont des objets différents.
- Pour les objets immuables, on peut se permettre d'avoir un seul objet, ce qui économise de l'espace mémoire.
- C'est donc la mutabilité d'un objet qui détermine ce comportement.

## Alias

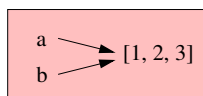
Dans certains cas, on veut que deux variables réfèrent le même objet même s'il est mutable.

On appelle cela un *alias* : il suffit d'assigner la première variable à la seconde (plutôt qu'une valeur équivalente).

```

>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True

```



Modifier un objet qui est référé par plusieurs variables est visible depuis l'ensemble de celles-ci.

```

>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> c = b
>>> print(a, b, c)
[1, 2, 3] [1, 2, 3] [1, 2, 3]
>>> a is b, a is c, b is c
(False, False, True)
>>> a[0] = 4
>>> print(a, b, c)
[4, 2, 3] [1, 2, 3] [1, 2, 3]
>>> b[0] = 5
>>> print(a, b, c)
[4, 2, 3] [5, 2, 3] [5, 2, 3]

```

Exercice : utilisez Python Tutor pour explorer comment sont représentés a, b et c

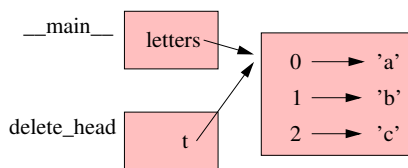


### Objets mutables en arguments

Quand on passe une liste en argument d'une fonction, la fonction reçoit une *référence* vers la liste : le paramètre de la liste est un alias et les modifications de la liste sont visibles en dehors de la fonction.

```
>>> def delete_head(t):
 t.pop(0)

>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
```



(voir code `delete_head` dans Python Tutor)

La fonction ci-dessus a-t-elle un effet de bord ?

Rappel : cf. Chapitre 3.

**Définition.** On dit qu'une fonction possède un *effet de bord* si celle-ci produit un effet qui est visible en dehors de la fonction. Le fait de retourner une valeur n'est pas considéré comme un effet de bord.

Donc, oui, une fonction qui modifie un objet mutable passé en argument possède un effet de bord.

Comment écrire une fonction qui ferait le même travail mais sans effet de bord ?

Puisqu'on ne peut pas modifier les paramètres, il faut *retourner* un nouvel objet (ici une nouvelle liste).

Voici une version sans effet de bord qui exploite le fait que l'opérateur `[]` retourne une nouvelle liste.

```
>>> def safe_del_head(t):
 return t[1:]

>>> letters = ['a', 'b', 'c']
>>> new_letters = safe_del_head(letters)
>>> print(letters)
['a', 'b', 'c']
>>> print(new_letters)
['b', 'c']
```

(voir code `safe_delete_head` dans Python Tutor)

Une fonction sans effet de bord peut aussi être utilisée pour mettre à jour une liste passée en paramètres, grâce à l'assignation.

```
>>> letters = ['a', 'b', 'c']
>>> letters = safe_del_head(letters)
>>> print(letters)
['b', 'c']
```

Elle offre donc le *contrôle* de choisir parmi les deux possibilités, contrairement à une fonction avec effet de bord.

Il est important de faire la distinction entre ce qui modifie un objet et ce qui en retourne

un nouveau.

Par exemple, la méthode `append` modifie la liste (effet de bord) (et retourne `None`).

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1, t2)
[1, 2, 3] None
```

Par contre, l'opérateur `+` retourne une nouvelle liste.

```
>>> t3 = t1 + [3]
>>> print(t1, t3)
[1, 2, 3] [1, 2, 3, 3]
```

En général :

- les *fonctions* sont *sans effet de bord* (par ex. `math.sqrt`)
- les *méthodes* sont
  - avec un *effet de bord* si l'objet est *mutable* (par ex. `list.append`, `list.pop`, `list.sort`)
  - sans *effet de bord* si l'objet est *immuable* (par ex. `str.upper`, `str.lower`)
- les *opérateurs* sont *sans effet de bord* (par ex. `+` et `[:]` pour les chaînes et les listes)

*Exercice.* Pourquoi la fonction suivante, censée supprimer le premier élément de la liste passée en paramètre, ne fonctionne pas ?

```
>>> def bad_delete_head(t):
 t = t[1:]

>>> letters = ['a', 'b', 'c']
>>> bad_delete_head(letters)
>>> print(letters)
['a', 'b', 'c']
```

- l'opérateur `[:]` crée une nouvelle liste et `t` réfère maintenant cette nouvelle liste
- le fait d'avoir changé la référence de `t` (intuition : la flèche sur les diagrammes d'état), ne signifie pas qu'on a touché à celle de `letters` (qui réfère toujours la liste de départ)

(voir code `bad_delete_head` dans Python Tutor)

## Liste de listes et copies

Avec des listes de listes, il y a quelques subtilités liées à la copie.

Que va afficher la dernière instruction ?

```
>>> A = [[0] * 3] * 2
>>> print(A)
[[0, 0, 0], [0, 0, 0]]
>>> A[0][0] = 1
>>> print(A)
```

On aimerait (et on s'attend à) obtenir

```
[[1, 0, 0], [0, 0, 0]]
```

mais on obtient

```
[[1, 0, 0], [1, 0, 0]]
```

Les deux "lignes" de cette matrice à 2 dimensions sont donc des alias → ce n'était sûre-

ment pas ce qui était souhaité!

Que va afficher ceci?

```
>>> A = [[1, 2], [3, 4]]
>>> B = A[:]
>>> A[0][0] = 9
>>> print(A, B)
```

On s'attend à ce que B soit une copie distincte de A comme ce serait le cas si A était une simple liste d'objets immuables :

```
[[9, 2], [3, 4]] [[1, 2], [3, 4]]
```

mais on obtient :

```
[[9, 2], [3, 4]] [[9, 2], [3, 4]]
```

*Explication* : les 2 situations précédentes sont dues au fait que l'on copie les *références* vers les objets mutables que sont les sous-listes. Les opérateurs \* et [:] ne réalisent pas une copie "en profondeur" : ils copient les références mais ne les suivent pas.

(Voir codes `matrix` dans Python Tutor)

*Solution* ? Utiliser la fonction `deepcopy` du module `copy`. Celle-ci va "suivre" les références pour effectuer une copie réellement distincte (quel que soit le nombre ou la profondeur des références à suivre).

```
>>> import copy
>>> A = [[1, 2], [3, 4]]
>>> B = copy.deepcopy(A)
>>> A[0][0] = 9
>>> print(A, B)
[[9, 2], [3, 4]] [[1, 2], [3, 4]]
```

## 7.6. Arguments de la ligne de commande (parenthèse)

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* :

- la commande `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 `myscript.py`.

Le nom du script est un argument de la commande `python3` qui lance l'interpréteur.

Tous les arguments passés après la commande `python3` (et séparés par des espaces<sup>3</sup>) sont disponibles via l'attribut `argv` du module `sys`. La valeur `argv` est une liste qui contient ces arguments.

Fichier `args.py` :

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

La commande "`python3 args.py`" produit :

---

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

```
['args.py']
```

La commande “python3 args.py un deux trois "x + y"” produit :

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

Amélioration du script qui trie les mots d’un fichier :

```
...
import sys

filename = ''
if len(sys.argv) < 2:
 filename = input('Nom du fichier: ')
else:
 filename = sys.argv[1]

with open(filename) as file:
 ...
```

## 7.7. Erreurs fréquentes et debug

Une utilisation non rigoureuse des listes (ou d’autres objets mutables) peut mener à des problèmes difficiles à déboguer.

- la plupart des méthodes sur les listes modifient celles-ci (et retournent `None`). C’est l’inverse pour les méthodes sur les chaînes (qui ne pourraient pas modifier l’argument) : elles retournent une nouvelle chaîne.

*Exemple :*

```
word = word.strip()
t = t.sort() # FAUX: t refere None !
```

- il faut donc lire avec attention la documentation des fonctions, méthodes et opérateurs ; et écrire votre propre documentation de manière claire. Le mode interactif est très utile pour tester rapidement l’utilisation d’une fonction ou d’une méthode.

- faites une copie si nécessaire.

```
import copy
t_original = copy.deepcopy(t)
t.sort()
```

- *Choisissez une manière de faire, apprenez-la correctement.*

Avec les listes  $\exists$  de nombreuses manières de faire la même chose :

- pour supprimer un élément d’une liste : `pop`, `remove` ou même `[x:y]`.
- pour ajouter un élément d’une liste : `append` ou opérateur `+`

Ces instructions sont correctes pour ajouter un élément :

```
t.append(x)
t = t + [x]
```

Ces instructions sont erronées :

```
t.append([x])
t = t.append(x)
t + [x]
t = t + x
```

*Exercice :* essayez ces exemples interactivement pour comprendre ce qu’ils font (seule la dernière instruction provoque une exception).

## 7.8. Exercices interactifs

### Exercices interactifs

*Exercice.* Ecrire un programme qui affiche les paires de mots (contenus dans le fichier `words.txt`) qui sont l'inverse l'un de l'autre. Par exemple, `stop` et `pots`.

*Exercice.* Ecrire un programme qui estime la probabilité que  $k \geq 2$  étudiants aient leur anniversaire le même jour au sein d'une classe de  $n$  étudiants. Les paramètres  $n$  et  $k$  sont passés en ligne de commande.

*Illustration :* découverte du code de l'énigme de la traversée.

## 7.9. Pour aller plus loin (à lire par soi-même)

### A propos des listes dans une boucle `for`

Pour accéder aux indices et aux valeurs d'une liste, on peut utiliser `enumerate` :

```
>>> t = [3, 5, 2, 99]
>>> for i, x in enumerate(t):
 print(i, x)
```

```
0 3
1 5
2 2
3 99
```

Une boucle `for` sur une liste vide n'exécute jamais son corps.

```
for x in []:
 print('This never happens.')
```

### Exemple d'utilisation des opérateurs sur les listes

Comment construire la liste suivante ?

[1, 3, 0, 0, 9, 11, 13, 15, 1, 3, 0, 0, 9, 11, 13, 15]

```
>>> liste = list(range(1, 16, 2))
>>> print(liste)
[1, 3, 5, 7, 9, 11, 13, 15]
>>> liste[2:4] = [0] * 2
>>> print(liste)
[1, 3, 0, 0, 9, 11, 13, 15]
>>> liste = liste * 2
>>> print(liste)
[1, 3, 0, 0, 9, 11, 13, 15, 1, 3, 0, 0, 9, 11, 13, 15]
```

### D'autres méthodes utiles sur les listes

La méthode `extend` prend une liste en argument et ajoute tous ses éléments en fin de la liste.

```
>>> help(list.extend)
Help on method_descriptor:
```

```

extend(...)
 L.extend(iterable) -> None -- extend list by appending elements
 from the iterable

>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']

```

La méthode `count` compte toutes les occurrences d'une valeur passée en argument.

```

>>> help(list.count)
Help on method_descriptor:

count(...)
 L.count(value) -> integer -- return number of occurrences of value

>>> t = ['a', 'b', 'c', 'a']
>>> t.count('a')
2

```

La méthode `insert` permet d'insérer un élément à un endroit donné de la liste.

```

>>> help(list.insert)
Help on method_descriptor:

insert(...)
 L.insert(index, object) -- insert object before index

>>> t = ['a', 'b', 'c']
>>> t.insert(0, 'z')
>>> print(t)
['z', 'a', 'b', 'c']
>>> t.insert(2, 'x')
>>> print(t)
['z', 'a', 'x', 'b', 'c']

```

La méthode `remove` supprime la première occurrence d'une valeur donnée.

```

>>> help(list.remove)
Help on method_descriptor:

remove(...)
 L.remove(value) -> None -- remove first occurrence of value.
 Raises ValueError if the value is not present.

>>> t = ['a', 'b', 'c', 'b']
>>> t.remove('b')
>>> print(t)
['a', 'c', 'b']
>>> t.remove('z')

Traceback (most recent call last):
 File "<pyshell#100>", line 1, in <module>
 t.remove('z')
ValueError: list.remove(x): x not in list

```

La méthode `index` retourne l'indice de la première occurrence d'une valeur donnée (des arguments optionnels permettent de limiter la recherche à un intervalle d'indices).

```

>>> help(list.index)
Help on method_descriptor:

index(...)
 L.index(value, [start, [stop]]) -> integer -- return first index of value.
 Raises ValueError if the value is not present.

```

```
>>> t = ['a', 'b', 'c', 'a']
>>> t.index('a')
0
>>> t.index('z')

Traceback (most recent call last):
 File "<pyshell#78>", line 1, in <module>
 t.index('z')
ValueError: list.index(x): x not in list
>>> t.index('a', 1)
3
```

## Supprimer les éléments d'une liste

Outre les méthodes `remove` et `pop`, l'opérateur `del` peut également être utilisé pour supprimer un élément à partir d'un indice (ou d'une tranche d'indices).

```
>>> t = ['a', 'b', 'c', 'd']
>>> x = t.pop(1)
>>> print(x)
b
>>> print(t)
['a', 'c', 'd']
>>> x = t.remove('c')
>>> print(x)
None
>>> print(t)
['a', 'd']
>>> del t[1]
>>> print(t)
['a']
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

## Listes et chaînes

Un `str` est une séquence de caractères et une liste est une séquence de valeurs, mais une liste de caractères n'est pas la même chose qu'une chaîne.

Pour convertir une chaîne en liste, on peut utiliser `list`.

```
>>> s = 'hello'
>>> t = list(s)
>>> print(t)
['h', 'e', 'l', 'l', 'o']
```

- Comme `list` est le nom d'une fonction, on ne peut pas l'utiliser comme nom de variable (mais `liste` peut l'être).
- En général, on évite d'utiliser la lettre `l` car elle ressemble trop au nombre `1`, c'est pourquoi `t` est souvent utilisé dans les exemples.

La fonction `list` sépare une chaîne en caractères individuellement. Nous avons vu la méthode `split` qui permet de séparer une chaîne en mots ou en sous-chaînes.

La méthode `join` fait le contraire de `split`. C'est une méthode sur les chaînes : on l'invoque sur le délimiteur et on passe la liste en paramètre (elle peut être utile après un `map`).

```
>>> s = 'to be or not to be'
```

```
>>> t = s.split()
>>> print(t)
['to', 'be', 'or', 'not', 'to', 'be']
>>> s = 'spam*spam*spam'
>>> delimiter = '*'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
>>> delimiter = ' '
>>> delimiter.join(t)
'to be or not to be'
>>> ''.join(t)
'tobeornottobe'
```

## 7.10. Glossaire

*élément* : une des valeurs dans une liste (ou une autre séquence).

*indice* : valeur entière qui indique la position d'un élément dans une liste (ou une séquence).

*liste* : séquence de valeurs qui peuvent être modifiées et mises entre crochets droits.

*liste imbriquée* : liste qui est un élément d'une autre liste.

*accumulateur* : variable utilisée dans une boucle pour additionner ou accumuler un résultat.

*filtre* : opération qui consiste à traverser une liste (ou une séquence) et qui sélectionne les éléments satisfaisant un certain critère.

*mapping* : opération qui consiste à traverser une liste (ou une séquence) pour appliquer une action sur chaque élément.

*réduction* : opération qui consiste à traverser une liste (ou une séquence) et qui accumule les éléments en un seul résultat.

*object* : quelque chose qu'une variable peut référer. Un objet possède un type et une valeur. On peut lui appliquer des méthodes définies en fonction de son type.

*référence* : l'association entre une variable et sa valeur (c-à-d la valeur de son objet).

*alias* : assigner une variable depuis une autre, pour qu'elles réfèrent au même objet.

*équivalent* : ayant la même valeur.

*identique* : être le même objet (ce qui implique l'équivalence).



---

## Prouver les propriétés des algorithmes

*Une élection étrange • Preuve d'exactitude d'un algorithme itératif • Exemple 1 (boucle while) : la division euclidienne • Exemple 2 (boucle while) : calcul de l'exposant • Exemple 3 (liste et invariant graphique) : maximum d'une liste • Exemple 4 (boucle for) : somme d'une séquence • Exemple 5 (boucle for) : ajout des carrés • Exemple 6 (plusieurs boucles imbriquées) : tri par sélection*

---

### Preuves des propriétés des algorithmes

Etant donné un (morceau) d'algorithme, on désire prouver que si les *préconditions* sont respectées, alors :

- *l'algorithme s'arrête* (prouver qu'il n'y a pas de boucle ou de récursion infinie, que l'algorithme s'arrête toujours) ;
- *l'algorithme est exact* (prouver que l'algorithme fait bien ce qu'il est supposé faire, que les *postconditions* sont respectées).

Dans ce chapitre (et ce cours) :

- nous allons présenter quelques algorithmes et prouver leurs propriétés (arrêt et exactitude) ;
- on se limite aux algorithmes *itératifs*. Les preuves concernant les algorithmes *récurifs* seront couvertes dans le cours de *Structures de Données I*.

### 8.1. Une élection étrange

#### Election à Weirldand

Deux candidats se sont présentés à l'élection présidentielle de Weirldand : le candidat A et le candidat B. Les bulletins de vote ont été rassemblés dans une urne<sup>1</sup> et les assesseurs s'apprêtent à dépouiller les résultats selon la méthode en vigueur dans leur pays, définie comme suit.

---

1. On considère qu'un bulletin contient soit A, soit B et qu'il n'y a pas de bulletin blanc ou nul.

« Tant qu'il reste plus d'un seul bulletin de vote dans l'urne : prendre deux bulletins de vote au hasard dans l'urne ; si les deux bulletins tirés sont en faveur du même candidat, les jeter et ajouter un nouveau bulletin pour A (disponibles en quantité infinie) ; sinon (un bulletin A et un bulletin B), jeter le bulletin en faveur de A et remplacer le bulletin B dans l'urne. »

Durant cette procédure étrange, s'il ne reste qu'un seul bulletin, alors ce bulletin correspond au gagnant de l'élection.

La fonction `depouillement` (voir slide suivant) simule la procédure.

Elle prend en entrée une liste (Python) ne contenant que des 'A' ou des 'B' (précondition).

```
import random

def depouillement(urne):
 while len(urne) > 1:
 print('Urne au début de l\'itération:', urne, end=' ')
 random.shuffle(urne)
 print(' -> ', urne)
 vote1 = urne.pop()
 vote2 = urne.pop()
 print("Votes retirés :", vote1, vote2, end=' ')
 if vote1 == vote2:
 print(' (identiques: A ajouté)')
 urne.append('A')
 else:
 print(' (différents: B ajouté)')
 urne.append('B')
 print('Urne à la fin de l\'itération:', urne, '\n', ('-' * 30))
```

(La fonction `random.shuffle` permet de réordonner aléatoirement les éléments d'une liste passée en paramètre.)

Par exemple, voici une exécution possible de `depouillement(urne)` si `urne` référence la liste ['A', 'A', 'B', 'B', 'A'] :

```
Urne au début de l\'itération: ['A', 'A', 'B', 'B', 'A'] -> ['B', 'A', 'B', 'A', 'A']
Votes retirés : A A (identiques: A ajouté)
Urne à la fin de l\'itération: ['B', 'A', 'B', 'A']

Urne au début de l\'itération: ['B', 'A', 'B', 'A'] -> ['A', 'B', 'B', 'A']
Votes retirés : A B (différents: B ajouté)
Urne à la fin de l\'itération: ['A', 'B', 'B']

Urne au début de l\'itération: ['A', 'B', 'B'] -> ['A', 'B', 'B']
Votes retirés : B B (identiques: A ajouté)
Urne à la fin de l\'itération: ['A', 'A']

Urne au début de l\'itération: ['A', 'A'] -> ['A', 'A']
Votes retirés : A A (identiques: A ajouté)
Urne à la fin de l\'itération: ['A']
```

Selon cette procédure, A serait donc déclaré vainqueur.

Les habitants de Weirdland prétendent que ce type de dépouillement se termine toujours.

De plus, et bien que la procédure est en partie aléatoire, certains affirment qu'il existe une règle déterministe permettant de connaître le candidat qui est déclaré vainqueur. La règle serait la suivante :

« Si le nombre de bulletins pour B est pair, alors A est élu. Sinon, B est élu. » (★)

Si on suppose que le paramètre `urne` référence une liste non vide ne contenant que des 'A' et des 'B', est-il possible de prouver que

a) la fonction `depouillement` se termine toujours ?

- b) si la liste `urne` contient initialement  $x$  'B' et que si l'on note par  $y$  le nombre de 'B' courant dans la liste (entre 2 itérations de la boucle), alors l'assertion

$$S \equiv x \text{ et } y \text{ ont la même parité}$$

est vraie

- *avant* d'entrer dans la boucle ; et
  - *après* chaque itération *complète* de la boucle ?
- c) si les assertions ci-dessus sont prouvées, alors la règle ( $\star$ ), énoncée par les habitants de Weirdland, est toujours vraie ?

a) La fonction `depouillement` se termine-t-elle toujours ?

```
def depouillement(urne):
 while len(urne) > 1:
 ...
 vote1 = urne.pop()
 vote2 = urne.pop()
 print("Votes retirés :", vote1, vote2, end=' ')
 if vote1 == vote2:
 print(' (identiques: A ajouté)')
 urne.append('A')
 else:
 print(' (différents: B ajouté)')
 urne.append('B')
```

- A chaque itération, deux votes sont retirés et, quelque soit le résultat du `if`, un vote est rajouté ;
- Il y a donc exactement un bulletin en moins à chaque itération ;
- La boucle s'arrêtera quand il n'y aura plus qu'un bulletin.

b) L'assertion " $S \equiv x$  et  $y$  ont la même parité" est-elle vraie avant et après chaque itération ?

```
def depouillement(urne):
 while len(urne) > 1:
 ...
 vote1 = urne.pop()
 vote2 = urne.pop()
 print("Votes retirés :", vote1, vote2, end=' ')
 if vote1 == vote2:
 print(' (identiques: A ajouté)')
 urne.append('A')
 else:
 print(' (différents: B ajouté)')
 urne.append('B')
```

- Avant d'entrer dans la boucle,  $x = y$  puisque rien n'a été modifié ( $x$  est le nombre de 'B' au départ et  $y$  le nombre de 'B' courant)
- Il y a trois cas possibles :
  - Si on tire deux 'B', alors on ajoute un 'A' et  $y$  est diminué de 2
  - Si on tire deux 'A', alors on ajoute un 'A' et  $y$  est inchangé
  - Si on tire un 'A' et un 'B', alors on ajoute un 'B' et  $y$  est inchangé

- Dans tous les cas l'assertion reste vraie si elle était vraie à l'itération précédente. Par induction,  $S$  est vraie.

c) Grâce à a) et b), peut-on affirmer que la règle  $(\star)$ , énoncée par les habitants de Weirldland, est toujours vraie ?

« Si le nombre de bulletins pour  $B$  est pair, alors  $A$  est élu. Sinon,  $B$  est élu. »  $(\star)$

- On sait grâce à a) que la procédure s'arrête quand il n'y a plus qu'un seul bulletin ;
- On sait grâce à b) que s'il reste un seul bulletin pour ' $B$ ' à la fin, et donc un nombre impair, cela signifie qu'il y avait un nombre impair de bulletins pour ' $B$ ' au départ ;
- Ces deux éléments suffisent pour conclure que  $(\star)$  est vraie.

## 8.2. Preuve d'exactitude d'un algorithme itératif

### Preuve d'exactitude d'un algorithme itératif

La preuve complète qu'un algorithme itératif est exact revient à :

- montrer qu'il s'arrête ;
- à exhiber, pour chaque boucle, une propriété (*invariant de boucle*) qui, si elle est valide avant l'exécution d'un tour de boucle, est aussi valide après l'exécution du tour de boucle ; et la prouver
- vérifier que les conditions initiales rendent la propriété vraie en entrée du premier tour de boucle
- conclure que cette propriété est vraie en sortie du dernier tour de boucle ; un bon choix de la propriété prouvera qu'on a bien produit le résultat souhaité.

La principale difficulté de ce type de preuve réside dans la détermination de l'invariant de boucle.

### Preuve d'arrêt d'un algorithme itératif

Pour prouver qu'un algorithme itératif s'arrête (si les préconditions sont respectées), il faut prouver que *chaque boucle se termine en un nombre fini d'itérations*.

*Remarque* : c'est en général facile à montrer (voir par ex. preuve pour `countdown` au ch. 6) mais il y a certains rares cas où cela peut être très difficile (voir chap. 6 et la conjecture de Syracuse).

### Invariant de boucle

Intuitivement, un *invariant de boucle* est une assertion (une propriété) qui

- est vraie avant l'entrée dans la boucle ;
- est toujours vraie après chaque itération de la boucle.

*Remarques* :

- cela implique que si une boucle s'arrête, l'invariant est toujours vrai après l'exécution complète de celle-ci ;
- l'invariant ne doit pas être vérifié à tout moment dans le corps de la boucle, mais à la fin de chaque itération de celle-ci.

Plus formellement, on désire qu'une propriété  $\mathcal{P}$  soit vérifiée à la fin de l'exécution d'une boucle.

**Définition.** Un *invariant de boucle* est une propriété  $Q$  qui est

- vraie *avant* d'entrer dans la boucle (avant le 1er passage);
- reste vraie *après chaque* passage dans la boucle;
- vaut la propriété désirée  $\mathcal{P}$  à la *fin* de l'exécution complète de la boucle.

## 8.3. Exemple 1 (boucle while) : la division euclidienne

### Algorithme : division euclidienne

*Problème.* Soit  $a$  et  $b$  deux entiers tels que  $a \geq 0$  et  $b > 0$ . Comment déterminer le quotient (entier) et le reste de la division de  $a$  par  $b$ ?

*Idée de l'algorithme de division euclidienne :*

Initialiser le reste à  $a$ . Ensuite, soustraire  $b$  au reste tant que celui-ci est plus grand que  $b$ . Le nombre de soustractions est égal au quotient.

*Exemple :* Soit  $a = 14$  et  $b = 5$ .

| # soustractions | reste |
|-----------------|-------|
| 0               | 14    |
| 1               | 9     |
| 2               | 4     |

Le quotient de  $a$  par  $b$  est 2 et le reste est 4.

```

1 def division(a, b):
2 r = a
3 q = 0
4 while r >= b:
5 r = r - b
6 q = q + 1
7 return (q, r)

>>> division(14, 5)
(2, 4)

```

*A prouver :* si  $a$  et  $b$  sont deux entiers tels que  $a \geq 0$  et  $b > 0$ ,

- la fonction `division` s'arrête-t-elle toujours? (preuve de l'*arrêt*)
- après l'exécution de la fonction `division` les valeurs retournées correspondent-elles bien au quotient de  $a$  par  $b$  et au reste? (preuve de l'*exactitude*)

### Division euclidienne : preuve d'arrêt

```

1 def division(a, b):
2 r = a
3 q = 0
4 while r >= b:
5 r = r - b
6 q = q + 1
7 return (q, r)

```

*Preuve d'arrêt.* Soient  $a \geq 0$  et  $b > 0$ .

- *Base.* Soit  $a < b$ . Dans ce cas, comme  $r$  est initialisé à  $a$ , le test (4) de la boucle est faux et la boucle n'est jamais exécutée.

- Gén. Soit  $a \geq b$ .
  - Par récurrence, supposons que la boucle s'arrête pour tout  $r \leq k$  ( $k \geq 0$ ).
  - La boucle s'arrête également pour  $r = k + 1$  car l'instruction (5) diminue strictement la valeur de  $r$  (car  $b > 0$  et  $b$  n'est jamais modifié dans la boucle).  $\square$

*Remarque* : notez l'importance de l'hypothèse  $b > 0$  dans cette preuve!

### Division euclidienne : invariant de boucle

Nous allons montrer que l'assertion suivante est un *invariant* de la boucle `while` :

$$a = b \times q + r.$$

```

1 def division(a, b):
2 r = a
3 q = 0
4 while r >= b:
5 r = r - b
6 q = q + 1
7 return (q, r)

```

| $a$ | $b$ | $q$ | $r$ | $b \times q + r$  |
|-----|-----|-----|-----|-------------------|
| 14  | 5   | 0   | 14  | $5 \times 0 + 14$ |
| 14  | 5   | 1   | 9   | $5 \times 1 + 9$  |
| 14  | 5   | 2   | 4   | $5 \times 2 + 4$  |

$a = b \times q + r$  est-il un invariant de boucle ?

- Avant d'entrer dans la boucle :  $q$  est initialisé à 0 et  $r$  à  $a$ , donc  $b \times q + r = b \times 0 + a = a$  et l'assertion est vraie.
- il reste à prouver que cela reste toujours vrai *après* une itération de la boucle.

Considérons une itération donnée de la boucle et vérifions que l'assertion reste vraie à la fin de cette itération.

Les variables  $q$  et  $r$  étant modifiées dans la boucle, on peut utiliser les notations suivantes :

- $q_0$  et  $r_0$  : valeurs de  $q$  et  $r$  au début de l'itération ;
- $q_1$  et  $r_1$  : valeurs de  $q$  et  $r$  à la fin de l'itération.

Prouvons que l'assertion est toujours vraie après une itération donnée :

- Par hypothèse d'induction :  $a = b \times q_0 + r_0$
- On a  $r_0 = r_1 + b$  et  $q_0 = q_1 - 1$  (instructions 5 et 6)
- Donc,  $a = b \times q_0 + r_0 = b \times (q_1 - 1) + r_1 + b$
- Après simplification, on obtient  $a = b \times q_1 + r_1$
- L'assertion reste donc vraie après l'itération considérée et  $a = b \times q + r$  est bien un invariant de la boucle `while`  $\square$

La preuve *complète* de l'exactitude de la fonction `division` est donc :

- preuve de l'arrêt (voir ci-avant)
- preuve que  $a = b \times q + r$  est un invariant de la boucle (voir ci-avant)
- le résultat retourné est correct car :
  - la boucle se termine quand la condition est fausse, c-à-d, quand  $r < b$  (= négation de la condition);

- à ce moment là,  $a = b \times q + r$  est vrai, donc,  $q = \frac{a-r}{b}$  et  $r < b$ , ce qui correspond à la définition du quotient et du reste d'une division<sup>2</sup>.

## 8.4. Exemple 2 (boucle while) : calcul de l'exposant

### Algorithme : calcul de l'exposant

*Problème.* Soit  $a$  un nombre réel et  $n$  un entier  $\geq 0$ . Comment calculer  $a^n$  ?

- *Approche classique* : utiliser une boucle pour calculer  $a \times a \times \dots \times a$ , ce qui revient à faire  $n - 1$  multiplications
- *Exponentielle rapide* : exploiter le fait que
  - si  $n$  est pair, alors  $a^n = (a^2)^{\frac{n}{2}}$
  - si  $n$  est impair, alors  $a^n = a \cdot a^{n-1}$

*Exemple* :  $a^9 = a \cdot a^8 = a \cdot (a^2)^4 = a \cdot ((a^2)^2)^2$ , ce qui revient à 4 multiplications au lieu de 8

On verra au chapitre 9 que la deuxième version est effectivement (beaucoup) plus rapide. Mais ici, dans ce chapitre, montrons déjà que l'algorithme va effectivement nous retourner  $a^n$ .

```

1 def expo(a, n):
2 r = 1
3 b = a
4 i = n
5 while i > 0:
6 if i % 2 == 0:
7 b = b * b
8 i = i // 2
9 else:
10 r = r * b
11 i = i - 1
12 return r

>>> expo(3, 3)
27
>>> expo(2, 0)
1

```

### Division euclidienne : preuve d'arrêt

```

1 def expo(a, n):
2 r = 1
3 b = a
4 i = n
5 while i > 0:
6 if i % 2 == 0:
7 b = b * b
8 i = i // 2
9 else:
10 r = r * b
11 i = i - 1
12 return r

```

*Preuve d'arrêt.* Soit  $n \geq 0$ . Par récurrence sur  $n$ .

- *Base.* Si  $n = 0$ , la boucle n'est pas exécutée.

---

2. Ici nous utilisons le fait (sans le prouver, cf. cours de *Mathématiques élémentaires*) qu'il existe un couple unique d'entiers  $q$  et  $r$  tels que  $q = \frac{a-r}{b}$  et  $r < b$ .

- Gén. Soit  $n > 0$ 
  - Par récurrence, supposons que la boucle s'arrête pour tout  $i \leq n - 1$ ;
  - Si  $i$  est pair, l'instruction (8) diminue strictement sa valeur;
  - Si  $i$  est impair, l'instruction (11) diminue strictement sa valeur;
  - Par induction, la boucle s'arrête donc également quand  $i = n$ .

□

Nous allons montrer que l'assertion suivante est un *invariant* de la boucle `while` :

$$a^n = r \cdot b^i.$$

```

1 def expo(a, n):
2 r = 1
3 b = a
4 i = n
5 while i > 0:
6 if i % 2 == 0:
7 b = b * b
8 i = i // 2
9 else:
10 r = r * b
11 i = i - 1
12 return r

```

Soit  $a = 2$  et  $n = 9$ , alors  $a^n = 512$ .

| $r$ | $b$ | $i$ | $r \times b^i$           |
|-----|-----|-----|--------------------------|
| 1   | 2   | 9   | $1 \times 2^9 = 512$     |
| 2   | 2   | 8   | $2 \times 2^8 = 512$     |
| 2   | 4   | 4   | $2 \times 4^4 = 512$     |
| 2   | 16  | 2   | $2 \times 16^2 = 512$    |
| 2   | 256 | 1   | $2 \times 256^1 = 512$   |
| 512 | 256 | 0   | $512 \times 256^0 = 512$ |

$a^n = r \cdot b^i$  est-il un invariant de boucle ?

- Avant d'entrer dans la boucle : l'initialisation des variables (instr. 2 à 4) donne  $r \cdot b^i = 1 \cdot a^n = a^n$  et l'assertion est vraie.
- il reste à prouver que cela reste toujours vrai *après* une itération de la boucle.

Considérons une itération donnée de la boucle et vérifions que l'assertion reste vraie à la fin de cette itération.

Les variables  $a$  et  $n$  ne sont pas modifiées par la boucle. On va utiliser les notations  $i_0$ ,  $b_0$  et  $r_0$  pour les valeurs des variables  $i$ ,  $b$  et  $r$  au début de l'itération et  $i_1$ ,  $b_1$  et  $r_1$  pour leurs valeurs à la fin de l'itération.

```

1 def expo(a, n):
2 r = 1
3 b = a
4 i = n
5 while i > 0:
6 if i % 2 == 0:
7 b = b * b
8 i = i // 2
9 else:
10 r = r * b
11 i = i - 1
12 return r

```



Prouvons que l'assertion est toujours vraie après une itération donnée :

- Par hypothèse d'induction :  $a^n = r_0 \cdot b_0^{i_0}$
- Si  $i_0$  est pair, alors
  - les instr. 7 et 8 donnent  $b_0 = \sqrt{b_1}$ ,  $i_0 = 2i_1$  et  $r_0 = r_1$
  - Donc,  $a^n = r_0 \cdot b_0^{i_0} = r_1 \cdot \sqrt{b_1}^{2i_1} = r_1 \cdot b_1^{\frac{1}{2} \cdot 2i_1} = r_1 \cdot b_1^{i_1}$ .
- Si  $i_0$  est impair, alors
  - les instr. 10 et 11 donnent  $b_0 = b_1$ ,  $i_0 = i_1 + 1$  et  $r_0 = \frac{r_1}{b_1}$
  - Donc,  $a^n = r_0 \cdot b_0^{i_0} = \frac{r_1}{b_1} \cdot b_1^{(i_1+1)} = \frac{r_1}{b_1} \cdot b_1 \cdot b_1^{i_1} = r_1 \cdot b_1^{i_1}$ .
- Quelle que soit la parité de  $i_0$ , l'assertion reste vraie après l'itération considérée et  $a^n = r \cdot b^i$  est donc bien un invariant de la boucle  $\square$

La preuve *complète* de l'*exactitude* de la fonction `expo` est donc :

- preuve de l'arrêt (voir ci-avant);
- preuve que  $a^n = r \cdot b^i$  est un invariant de la boucle (voir ci-avant);
- le résultat retourné (valeur de  $r$ ) est correct car :
  - quand la boucle se termine, on a  $i = 0$ ;
  - après la boucle, l'invariant devient donc :

$$r = \frac{a^n}{b^i} = \frac{a^n}{b^0} = \frac{a^n}{1} = a^n;$$

- la valeur retournée est donc bien  $a^n$ .

## 8.5. Exemple 3 (liste et invariant graphique) : maximum d'une liste

### Algorithme de parcours d'une liste : maximum d'une liste

La notation indicée pour les variables modifiées dans une boucle n'est pas toujours bien adaptée à certains invariants.

*Exemple* : connaître la valeur maximum contenue dans une liste d'entiers.

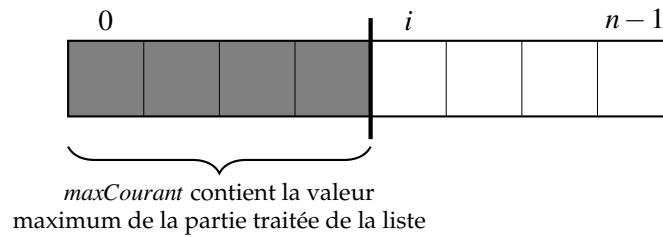
```
def maximum(A):
 if len(A) == 0:
 return None
 maxCourant = A[0]
 i = 1
 n = len(A)
 while i < n:
 if A[i] > maxCourant:
 maxCourant = A[i]
 i = i + 1
 return maxCourant

max = maximum([7, 8, 4, 2, 65, 7, 3, 17])
print('La plus grande valeur est', max)
```

### Preuve graphique

Dans cet exemple (et pour beaucoup d'algorithmes parcourant une liste), il est plus pratique de *représenter l'invariant graphiquement* et d'en expliquer la preuve directement via ce *dessin*.

*Invariant (propriété Q) :*

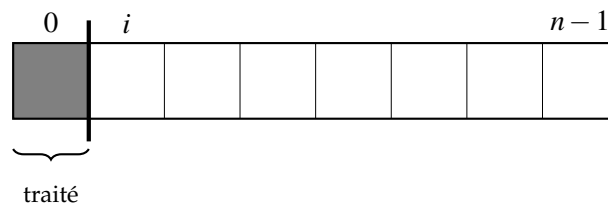


### Avant d'entrer dans la boucle

```

maxCourant = A[0]
i = 1
n = len(A)
...

```



*maxCourant* contient bien la valeur maximum de la partie traitée de la liste.

### Après chaque nouvelle itération

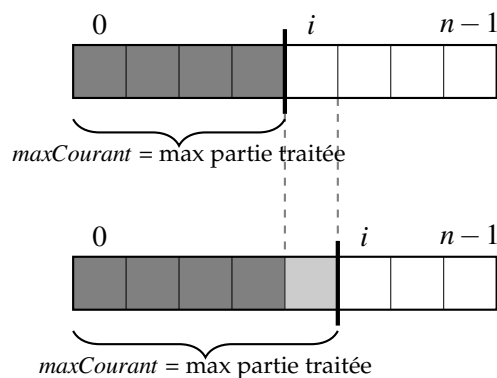
On suppose l'invariant vrai après  $k$  itérations (ici  $k = i - 1$ ).

Cette hypothèse, ainsi que le corps de la boucle, assurent qu'après l'itération suivante,  $Q$  est toujours vérifiée.

```

...
while i < n:
 if A[i] > maxCourant:
 maxCourant = A[i]
 i = i + 1
...

```



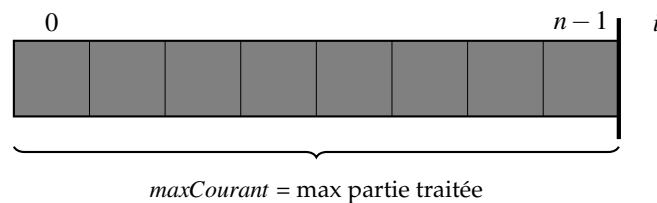
### Après l'exécution complète de la boucle

```

...
while i < n:
 ...
return maxCourant

```

A la fin,  $i = n$  et la propriété  $Q$  vérifiée précédemment devient :



Ceci conduit à la propriété  $\mathcal{P}$  désirée.

## 8.6. Exemple 4 (boucle for) : somme d'une séquence

### Algorithme avec une boucle for : preuve d'arrêt

Prouver qu'une boucle `for` s'arrête est très simple si celle-ci est appliquée sur une séquence respectant les 3 conditions suivantes :

- la boucle itère sur chaque élément de la séquence (une et une seule fois);
- la séquence est finie;
- le corps de la boucle ne modifie pas le nombre d'éléments de la séquence.

En effet, sous ces hypothèses il est évident que la boucle se termine en un nombre fini d'étapes.

### Somme d'une séquence : preuve d'arrêt

En Python, les deux premières conditions sont automatiquement gérées par la syntaxe d'une boucle `for` et par le fait qu'une séquence est toujours finie.

Donc, si le corps de la boucle ne modifie pas (n'augmente pas) le nombre d'éléments de la séquence, la preuve de l'arrêt est immédiate.

```

1 def sum_all(liste):
2 res = 0
3 for item in liste:
4 res += item
5 return res

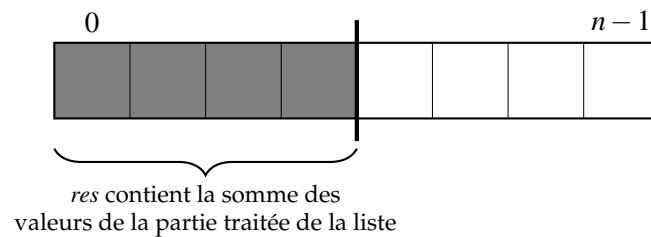
```

*Remarque* : pour le moment nous travaillons avec des séquences finies, donc ce qui est dit ici est correct, *mais* il existe une notion de *générateurs* qui permettent de générer des séquences infinies. Dans ce cas, la preuve d'arrêt doit être faite.

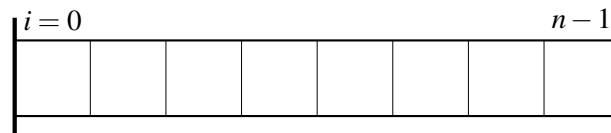
### Somme d'une séquence : invariant de boucle

Le fait qu'une boucle `for` s'applique linéairement sur une séquence rend l'utilisation d'une preuve graphique aisée.

*Invariant (propriété  $Q$ ) :*

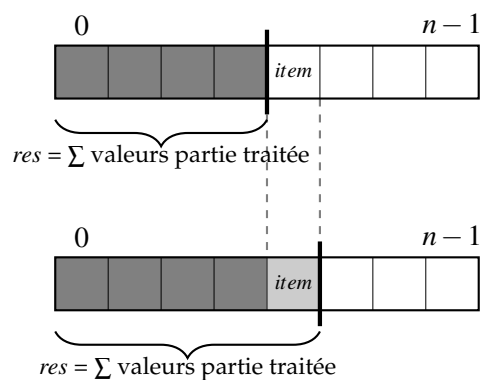


**Avant d'entrer dans la boucle**



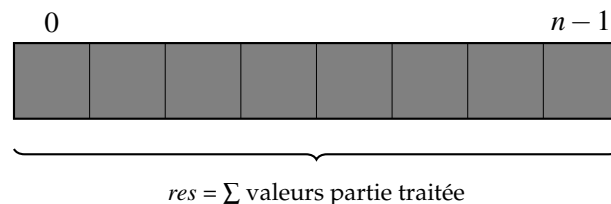
La partie traitée est vide, et l'invariant est donc trivialement vrai.

**Après chaque nouvelle itération**



**Après l'exécution complète de la boucle**

A la fin, la boucle `for` a parcouru toute la liste et la propriété  $Q$  vérifiée précédemment devient :



Ceci conduit à la propriété  $\mathcal{P}$  désirée.

## 8.7. Exemple 5 (boucle for) : ajout des carrés

La fonction suivante prend une liste d'entiers en argument *mais augmente la taille de la liste* en ajoutant après chaque élément son carré.

```

1 def add_square(liste):
2 index = 0
3 for item in liste:
4 if index % 2 == 0:
5 liste.insert(index+1, item*item)
6 index = index + 1

>>> t = [1, 2, 3]
>>> add_square(t)
>>> t
[1, 1, 2, 4, 3, 9]

```

Ici une preuve d'arrêt serait nécessaire, mais cette fonction peut être réécrite de telle sorte qu'une preuve est inutile (voir slide suivant). Ce genre de code est à éviter!

On construit une nouvelle liste et la preuve devient inutile. On retourne la liste modifiée plutôt que de la transformer dans la fonction.

```

1 def add_square(liste):
2 res = []
3 for item in liste:
4 res.extend([item, item * item])
5 return res

>>> t = [1, 2, 3]
>>> t = add_square(t)
>>> t
[1, 1, 2, 4, 3, 9]

```

La méthode `t1.extend(t2)` permet d'ajouter tous les éléments d'une liste `t2` à la fin de la liste `t1`.

## 8.8. Exemple 6 (plusieurs boucles imbriquées) : tri par sélection

### Algorithme : tri par sélection

*Problème.* Soit une séquence de  $n$  entiers ( $n \geq 0$ ), comment trier (par ordre croissant) les éléments de la séquence?

*Idée du tri par sélection.*

Imaginons que nous devons trier une main d'un jeu de cartes. On pose les cartes faces visibles sur la table. On sélectionne la plus "petite" carte et on la prend en main. On sélectionne ensuite la plus petite carte parmi celles restées sur la table et on la place à droite de la carte en main. On répète le processus jusqu'à avoir sélectionné toutes les cartes posées sur la table.



*Idée du tri par sélection.*

Pour appliquer cette idée à une séquence d'entiers, on représente la séquence comme suit : la partie grisée est la partie triée de la séquence (les cartes en main) et la partie

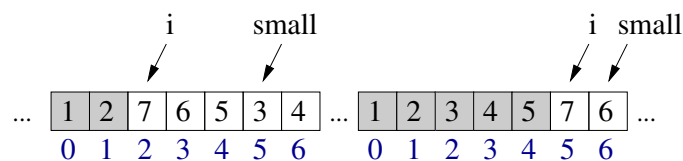
blanche représente la partie non triée (les cartes sur la table).

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 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 |

- sélectionner une valeur revient à échanger la première valeur non triée avec la plus petite valeur non triée (avec les cartes cela revient à les trier par échanges successifs en les laissant sur la table)
- quand il ne reste plus qu'un élément à trier, on peut s'arrêter

On va utiliser deux boucles imbriquées :

- Une première boucle permet de sélectionner chaque élément un par un : quand l'indice de la première boucle vaut  $i$ , les  $i$  premiers éléments sont triés (indices 0 à  $i-1$ ). A chaque itération de cette boucle, on déplace le plus petit élément de la partie non triée, pour élargir d'une case la partie triée.
- Une deuxième boucle (interne à la première) est donc nécessaire pour déterminer l'indice de la plus petite valeur des éléments non triés (*small*).



```

1 def selection_sort(t):
2 n = len(t)
3 for i in range(n-1):
4 small = i
5 for j in range(i+1, n):
6 if t[j] < t[small]:
7 small = j
8 (t[i], t[small]) = (t[small], t[i])

>>> t = [3, 7, 2, 6, 5, 1, 4]
>>> selection_sort(t)
>>> t
[1, 2, 3, 4, 5, 6, 7]
```

### Tri par sélection : preuve d'arrêt

S'il y a plusieurs boucles, il faut prouver que chacune d'elles s'arrête.

```

1 def selection_sort(t):
2 n = len(t)
3 for i in range(n-1):
4 small = i
5 for j in range(i+1, n):
6 if t[j] < t[small]:
7 small = j
8 (t[i], t[small]) = (t[small], t[i])
```

*Preuve.* On observe d'abord que, sous l'hypothèse qu'un indice  $i < n - 1$  est fixé, la boucle interne (instr. 5 – 7) s'arrête, car elle s'applique sur une séquence finie. Ensuite, on observe que la boucle externe (instr. 3 – 8) s'applique sur une séquence finie et s'arrête également.  $\square$

### Tri par sélection : invariants de boucles

```

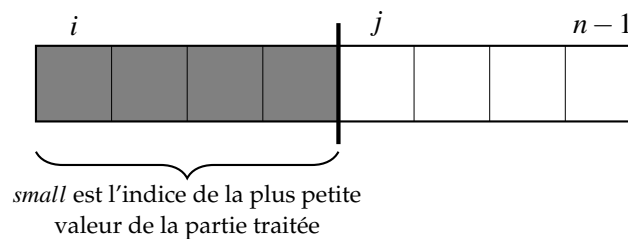
1 def selection_sort(t):
2 n = len(t)
3 for i in range(n-1):
4 small = i
5 for j in range(i+1, n):
6 if t[j] < t[small]:
7 small = j
8 (t[i], t[small]) = (t[small], t[i])

```

#### Invariants des boucles ?

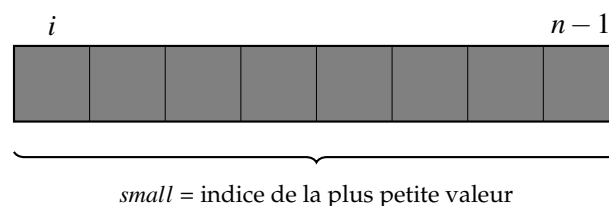
- *Boucle interne* : La valeur de `small` est l'indice du plus petit élément de la sous-séquence  $t[i], \dots, t[j-1]$
- *Boucle externe* : La partie traitée (indices de 0 à  $i-1$ ) est *triée* et toutes ses valeurs sont *inférieures ou égales* aux valeurs de la partie non-traitée.

#### Tri par sélection : invariant de la boucle interne

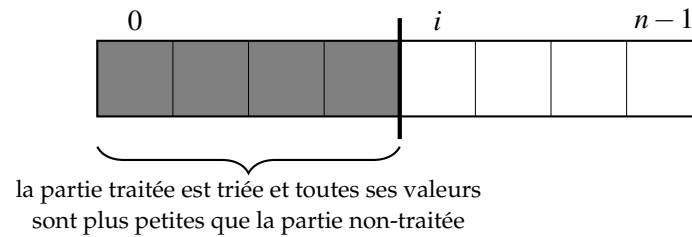


#### Tri par sélection : invariant de la boucle interne

A la fin de la boucle interne, on peut donc considérer que `small` contient l'indice de la plus petite valeur de la zone de la liste qui commence à  $i$  et va jusqu'à la fin.



#### Tri par sélection : invariant de la boucle externe

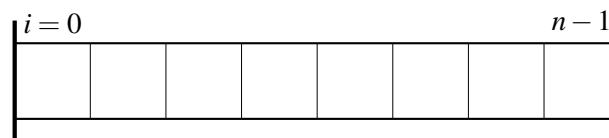


### Avant d'entrer dans la boucle externe

```

1 def selection_sort(t):
2 n = len(t)
3 for i in range(n-1):
4 small = i
5 for j in range(i+1, n):
6 if t[j] < t[small]:
7 small = j
8 (t[i], t[small]) = (t[small], t[i])

```



La partie traitée est vide, et l'invariant est donc trivialement vrai.

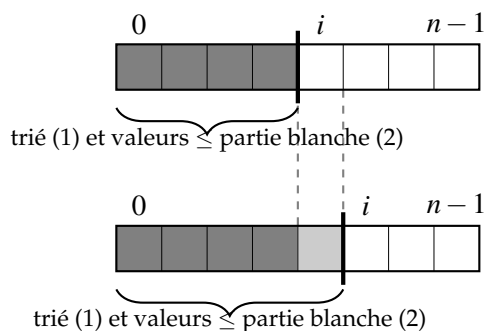
### Après chaque nouvelle itération de la boucle externe

```

1 def selection_sort(t):
2 n = len(t)
3 for i in range(n-1):
4 small = i
5 for j in range(i+1, n):
6 if t[j] < t[small]:
7 small = j
8 (t[i], t[small]) = (t[small], t[i])

```

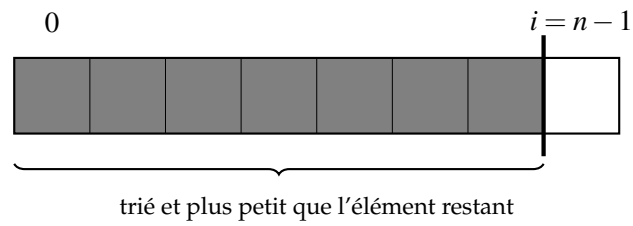
- on sait que la boucle interne nous donne l'indice (*small*) du plus petit élément de la partie blanche
- l'élément d'indice *small* est placé à l'indice *i*
- comme il est  $\geq$  que les valeurs déjà traitées, cela reste trié et (1) est vrai
- comme il est le plus petit de la zone blanche, (2) restera vrai



### Après l'exécution complète de la boucle externe

A la fin,  $i = n - 1$  et on se retrouve, grâce à l'invariant de la boucle externe, dans la configuration suivante :





Les données sont donc complètement triées et ceci conclut la preuve d'exactitude du tri par sélection.

---

## Complexité : évaluer l'efficacité des algorithmes

*Efficacité des algorithmes • Calculer le temps CPU des algorithmes • Quelques nouveaux algorithmes • La notation grand-O et la notion de complexité dans le pire des cas • Evaluer la complexité des algorithmes itératifs • Evaluer la complexité des algorithmes récursifs • Analyse du comportement de nos algorithmes • Glossaire*

---

### Référence

Le contenu de ce chapitre n'est pas couvert dans le livre *ThinkPython* de Downey.

En ce qui concerne la complexité, les références suivantes ont été utilisées :

- AHO, A. et ULLMAN, J., *Concepts fondamentaux de l'Informatique*, Dunod (1993)
- CORMEN, T. et LEISERSON, C., *Introduction to algorithms*, MIT Press (1991)

### 9.1. Efficacité des algorithmes

Etant donné un algorithme :

- comment évaluer son efficacité ?
- comment comparer l'efficacité de deux algorithmes différents résolvant le même problème ?

Pour cela, on peut :

- calculer le temps d'exécution utilisé par la machine pour effectuer un algorithme (on parle de temps *CPU*<sup>1</sup>) : temps utilisé par le processeur pour réaliser une tâche particulière ;
- évaluer l'efficacité théorique d'un algorithme, la croissance du temps mis par un algorithme en fonction de la taille de ses entrées : on parle de la *complexité* de l'algorithme.

L'efficacité d'un algorithme peut dépendre de la *taille* des entrées (par ex. nombre d'éléments dans une liste, nombre d'équations dans un système à résoudre, etc.).

---

1. Central Processing Unit.

Pour une taille  $n$  donnée, l'efficacité d'un algorithme (par ex. un tri) peut également dépendre des *valeurs* données en entrée. On peut alors déterminer :

- le meilleur des cas (par ex. les valeurs sont déjà triées);
- le pire des cas (par ex. les valeurs sont triées par ordre décroissant).

Ainsi, on parlera de l'efficacité ou de la complexité d'un algorithme

- dans le pire des cas;
- dans le meilleur des cas;
- en moyenne (entrées aléatoires).

## 9.2. Calculer le temps CPU des algorithmes

On ne calcule pas le temps d'exécution d'un algorithme avec un chronomètre!

En effet,

- le processeur n'est pas utilisé uniquement pour votre programme même si vous quittez toutes les autres applications. Il y a quantité de programmes qui tournent en tâches de fond (anti-virus, test de connexion internet, test pour savoir si un CD est dans le lecteur ou si un périphérique USB est branché, etc.)
- il y a aussi d'autres problèmes techniques difficiles à estimer : y-a-t il une utilisation de mémoire cache? Comment cela marche-t-il?

On va plutôt évaluer le *temps CPU* : temps dédié par le processeur pour exécuter uniquement un (morceau) de code donné.

*CPU = Central Processing Unit* (processeur)

Le temps CPU est dépendant du processeur, du système d'exploitation, etc. On ne peut donc comparer des tests que s'ils sont réalisés sur une même machine (avec la même configuration).

Pour estimer le temps CPU en Python, on utilise le module `timeit` qui règle automatiquement la plupart des problèmes évoqués ci-avant.

Néanmoins, cela reste une estimation, et le même morceau de code sur les mêmes entrées peut donner des temps différents.

C'est pourquoi il est recommandé de :

- lancer plusieurs séries de tests;
- pour chaque série, calculer la moyenne des temps CPU;
- ne garder que la *plus petite* de ces moyennes.

Voyez-vous pourquoi on prend la plus petite moyenne?

Car les différences de temps ne sont pas dues à votre code (s'il est déterministe), mais à d'autres facteurs!

### Utilisation du module `umons_cpu`

Pour faciliter et automatiser l'utilisation de `timeit`, nous avons écrit un module `umons_cpu` qui contient deux fonctions utiles :

- `cpu_time(f, *args)` pour calculer "comme il se doit" le temps CPU d'une fonction `f`;

- `calibrate(f, *args)` pour connaître le nombre de tests réalisés par `cputime` pour déterminer le temps CPU de `f`.

Le module `umons_cpu` et un script `cpu_examples` sont disponibles sur moodle.

```
>>> import umons_cpu
>>> help(umons_cpu.cpu_time)
Help on function cpu_time in module umons_cpu:

cpu_time(f, *args)
 Retourne un temps CPU exprimé en millisecondes (ms)
 - f : fonction ou méthode à tester
 - *args : liste d'arguments pour f. Ces arguments ne sont pas
 modifiés, même si la fonction f a des effets de bord (ils sont
 copiés avant l'exécution).

Exemples :
 cputime(math.sqrt, 4)
 pour calculer le temps CPU de math.sqrt(4)
 cputime(str.upper, 'hello')
 pour calculer le temps CPU de 'hello'.upper()
 cputime(myfunc, x, y, z)
 pour calculer le temps CPU de myfunc(x, y, z)
```

Soit le module `fibonacci.py` contenant les deux fonctions suivantes pour calculer le *n*ème nombre de Fibonacci :

```
def fib_rec(n):
 if n == 0:
 return 0
 elif n == 1:
 return 1
 else:
 return fib_rec(n-1) + fib_rec(n-2)

def fib_iter(n):
 f = [0, 1]
 for i in range(2, n+1):
 f.append(f[i-1] + f[i-2])
 return f[n]

>>> from umons_cpu import cpu_time
>>> from fibo import fib_iter, fib_rec
>>> fib_iter(30)
832040
>>> fib_rec(30)
832040
>>> cpu_time(fib_iter, 30)
0.009670206699956907
>>> cpu_time(fib_rec, 30)
481.5851666004164
>>> 481.58 / 0.00967
49801.447776628745
```

Calculer le 30ème nombre de Fibonacci

- prend moins d'un centième de milliseconde pour la version itérative ;
- prend environ une demi seconde pour la version récursive ;
- est donc environ 50000 fois plus lent avec la version récursive !

## Comparaison d'algorithmes

A la fin du module `fibonacci.py`, ajoutons :

```
if __name__ == '__main__':
 from umons_cpu import cpu_time
```

```

print("Temps affichés en msec")
print('n: iter: rec:')
for i in range (5, 31, 5):
 print('%4d ' % i, end=' ')
 print('%3f ' % (cpu_time(fib_iter, i)), end=' ')
 print('%3f' % (cpu_time(fib_rec, i)))
for i in range (100, 1901, 200):
 print('%4d ' % i, end=' ')
 print('%3f ----' % (cpu_time(fib_iter, i)))

```

```

Temps affichés en msec
n: iter: rec:
 5 0.003 0.004
 10 0.005 0.033
 15 0.006 0.353
 20 0.007 3.946
 25 0.008 43.773
 30 0.010 487.291
100 0.026 ----
300 0.075 ----
500 0.129 ----
700 0.190 ----
900 0.245 ----
1100 0.300 ----
1300 0.359 ----
1500 0.419 ----
1700 0.479 ----
1900 0.546 ----

```

Petite explication sur le fonctionnement de la fonction `cpu_time` :

- elle réalise 3 séries de  $n$  tests ;
- $n$  est une puissance de 10 déterminée automatiquement : elle vaut 10 au minimum, mais sera plus grande si les premiers tests sont rapides ;
- seule la plus rapide des 3 séries est utilisée ;
- le temps CPU retourné est le temps moyen d'un test, calculé pour la plus rapide des 3 séries.

```

>>> import umons_cpu
>>> help(umons_cpu.calibrate)
Help on function calibrate in module umons_cpu:

```

```

calibrate(f, *args)
 Retourne un nombre de tests qui rend le calcul du temps CPU
 a priori raisonnable.
 - f : fonction ou méthode à tester
 - *args : liste d'arguments pour f. Ces arguments ne sont pas
 modifiés, même si la fonction f a des effets de bord (ils sont
 copiés avant l'exécution).

```

Le nombre de tests retourné est une puissance de 10 (au minimum 10). Il sera d'autant plus grand si la fonction semble rapide.

```

>>> from umons_cpu import calibrate
>>> from fibo import fib_iter, fib_rec
>>> calibrate(fib_iter, 30)
10000
>>> calibrate(fib_rec, 30)
10

```

Il reste encore à comprendre *pourquoi* la version récursive est tellement plus lente que la version itérative. Une analyse de la *complexité* de ces algorithmes nous donnera la réponse.

La fonction `cpu_time` réalise une copie en profondeur des arguments de la fonction à

tester, pour éviter les effets de bord. Cette copie prend un certain temps qui est inclus dans le temps CPU retourné. Pour beaucoup d’algorithmes, ce temps est négligeable.

Si la fonction à tester n’a pas d’effet de bord et que son temps d’exécution est très rapide, et si les arguments de la fonction sont lourds à copier, il peut-être utile d’utiliser plutôt `cpu_time_without_copy`.

Nous en verrons un exemple d’utilisation au chapitre 10.

## 9.3. Quelques nouveaux algorithmes

### Algorithme : tri par insertion

*Problème.* Soit une séquence de  $n$  entiers ( $n \geq 0$ ), comment trier (par ordre croissant) les éléments de la séquence ?

*Idée du tri par insertion.*

Imaginons que nous devons trier une main d’un jeu de cartes. On pose les cartes non triées sur la table (faces cachées). On prend une première carte en main : elle forme une “sous-main” triée. On prend une deuxième carte sur la table et on l’insère dans la main de telle sorte que les deux cartes en main soient triées. On répète le processus jusqu’à avoir inséré toutes les cartes posées sur la table.



*Idée du tri par insertion.*

Pour appliquer cette idée à une séquence d’entiers, on représente la séquence comme suit : la partie grisée est la partie triée de la séquence.

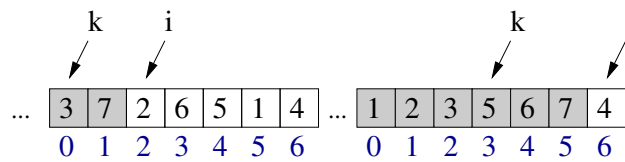
|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 3 | 7 | 2 | 6 | 5 | 1 | 4 |
| 3 | 7 | 2 | 6 | 5 | 1 | 4 |
| 2 | 3 | 7 | 6 | 5 | 1 | 4 |
| 2 | 3 | 6 | 7 | 5 | 1 | 4 |
| 2 | 3 | 5 | 6 | 7 | 1 | 4 |
| 1 | 2 | 3 | 5 | 6 | 7 | 4 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Deux boucles nécessaires :

- une boucle (indice  $i$ ) pour trier chaque élément 1 à 1 (sauf le premier)
- une boucle interne à la première (indice  $j$ ), pour décaler l’élément à trier vers la gauche jusqu’à sa place

*Exemple.* Quand l’indice de la première boucle vaut  $i$ , les  $i$  premiers éléments sont triés (indices 0 à  $i - 1$ ). Appelons *clef* la valeur de l’élément à trier. L’indice de la clef est  $i$ . Soit  $k$  la valeur de l’indice de la nouvelle position de la clef, après insertion : tous les éléments

de la partie triée dont les valeurs sont plus grandes que la clef ont un indice compris entre  $k$  et  $i - 1$ . On va utiliser une deuxième boucle d'indice  $j$  pour décaler ces éléments vers la droite.



Fonction en Python :

```

1 def insertion_sort(t):
2 n = len(t)
3 for i in range(1, n):
4 clef = t[i]
5 j = i - 1
6 while j >= 0 and t[j] > clef:
7 t[j+1] = t[j]
8 j = j - 1
9 t[j+1] = clef

>>> t = [3, 7, 2, 6, 5, 1, 4]
>>> insertion_sort(t)
>>> t
[1, 2, 3, 4, 5, 6, 7]
```

Remarque : à la ligne 9, le nouvel indice de la clef est  $j + 1$  car on a décrémenté  $j$  lors de la dernière itération de la boucle while.

### Algorithme : tri par fusion

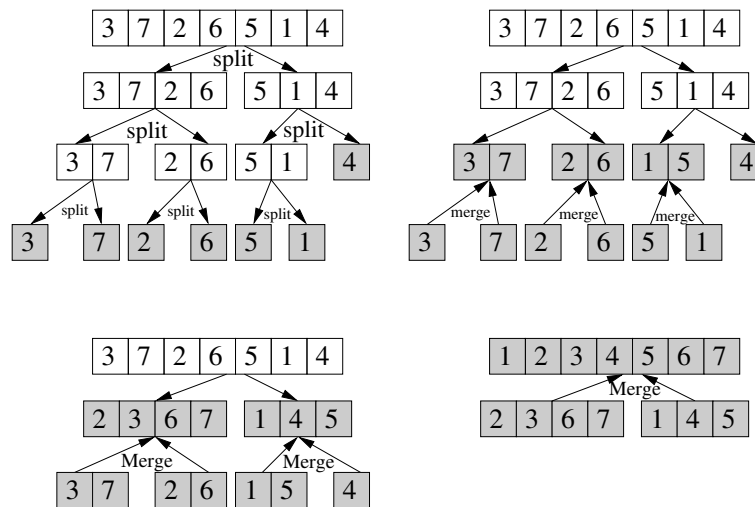
*Problème.* Soit une séquence de  $n$  entiers ( $n \geq 0$ ), comment trier (par ordre croissant) les éléments de la séquence ?

*Idée du tri par fusion.*

Imaginons que nous devons trier une main d'un jeu de cartes et que Bob soit en notre compagnie. On divise le paquet de cartes en 2 parties (à peu près) égales. On donne la première partie à Bob, qui s'occupe de la trier. On lui donne alors la seconde partie pour qu'il la trie également. Ensuite, à partir des deux paquets de cartes triées, on reconstitue un seul paquet de cartes trié en fusionnant les deux paquets triés.



Appliquons cette idée à une séquence d'entiers : on divise (split) et on fusionne deux parties triées (merge). Une séquence à 1 élément est triée. Les séquences triées sont grisées.



On va écrire

- une fonction `split` pour séparer une liste (approx.) en 2;
- une fonction `merge` qui fusionne deux listes triées;
- une fonction récursive `merge_sort`
  - *Base* : une liste vide ou à un seul élément est une liste triée;
  - *Gén.* : on divise la liste en 2, puis on trie récursivement les 2 parties, puis on fusionne les deux parties triées.
- contrairement aux tris par sélection et insertion qui ne modifient que les valeurs à l'intérieur de la liste<sup>2</sup>, ici nous devons travailler avec des nouvelles sous-listes. Pour éviter les problèmes (cf. Chap. 7), les fonctions retourneront les listes.

`insertion_sort(t)` # la liste est modifiée par la fonction  
`t = merge_sort(t)` # on retourne la liste triée

Fonction `split` en Python :

```
1 def split(t):
2 """ precondition: len(t) >= 2 """
3 mid = len(t) // 2
4 t1 = t[:mid]
5 t2 = t[mid:]
6 return (t1, t2)

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

Idée de l'algorithme `merge`.

On va écrire une fonction récursive `merge` qui fusionne deux séquences  $A_1$  et  $A_2$  triées :

- *Base* : si  $A_1$  est vide, alors  $A_2$  est la liste fusionnée; si  $A_2$  est vide, alors  $A_1$  est la liste fusionnée<sup>3</sup>
- *Gén.* : soit  $x$  le dernier élément de  $A_1$  et  $y$  le dernier élément de  $A_2$ . Si  $x > y$ , alors  $x$

2. On parle de fonctions réalisant le travail "in place".

3. Implique : si les deux listes sont vides, alors la liste fusionnée est vide.



doit se trouver en dernier dans la liste fusionnée<sup>4</sup>. La liste fusionnée est  $merge(A'_1, A_2)$  avec  $x$  ajouté à la fin et où  $A'_1$  est  $A_1$  pour laquelle on a retiré le dernier élément. Le cas  $x \leq y$  est géré de manière symétrique.

*Fonction merge en Python :*

```

1 def merge(t1, t2):
2 if len(t1) == 0:
3 return t2
4 elif len(t2) == 0:
5 return t1
6 elif t1[-1] > t2[-1]:
7 last = t1.pop()
8 new = merge(t1, t2)
9 new.append(last)
10 return new
11 else:
12 last = t2.pop()
13 new = merge(t1, t2)
14 new.append(last)
15 return new

>>> merge([], [])
[]
>>> merge([], [1])
[1]
>>> merge([1, 2, 7, 8], [3, 4, 9])
[1, 2, 3, 4, 7, 8, 9]
```

*Fonction merge\_sort en Python :*

```

1 def merge_sort(t):
2 n = len(t)
3 if n > 1:
4 (t1, t2) = split(t)
5 t1 = merge_sort(t1)
6 t2 = merge_sort(t2)
7 return merge(t1, t2)
8 else:
9 return t

>>> merge_sort(t)
[1, 2, 3, 4, 5, 6, 7]
>>> t
[7, 4, 2, 1, 6, 5, 3]
>>> t = merge_sort(t)
>>> t
[1, 2, 3, 4, 5, 6, 7]
```

## Efficacité des algorithmes de tri

*Questions :*

- comment exprimer l'efficacité du tri par sélection (cf. Chap. 8), du tri par insertion ou du tri par fusion en fonction du nombre  $n$  d'éléments dans la séquence à trier ?
- y-a-t il des cas (entrées) qui sont pires ou meilleurs que d'autres ?
- quel est, des trois algorithmes de tri présentés, le plus efficace ?

Dans le module `sort` :

```

from random import randint
from umons_cpu import cpu_time

def test(n):
 t1 = list(range(n))
 t2 = list(range(n, 0, -1))
 t3 = []
```

- 
4. On travaille en fin de liste plutôt qu'en début car on verra que c'est plus efficace un peu plus tard.

```

for i in range(n):
 t3.append(randint(0,n))
print('%6d ' % n, end=' ')
print('%7.2f' % (cpu_time(selection_sort, t1)), end=' ')
print('%7.2f' % (cpu_time(insertion_sort, t1)), end=' ')
print('%7.2f' % (cpu_time(merge_sort, t1)), end=' ')
print('%7.2f' % (cpu_time(selection_sort, t2)), end=' ')
print('%7.2f' % (cpu_time(insertion_sort, t2)), end=' ')
print('%7.2f' % (cpu_time(merge_sort, t2)), end=' ')
print('%7.2f' % (cpu_time(selection_sort, t3)), end=' ')
print('%7.2f' % (cpu_time(insertion_sort, t3)), end=' ')
print('%7.2f' % (cpu_time(merge_sort, t3)))

if __name__ == '__main__':
 print("Temps affichés en msec")
 print('n t1: sel ins mer t2: sel ins mer t3: sel ins mer')
 for i in range(100, 901, 100):
 test(i)

```

Comparaison des trois fonctions de tri ( $t_1$  déjà trié,  $t_2$  trié par ordre décroissant,  $t_3$  aléatoire) en fonction de la taille  $n$  de la séquence :

| Temps affichés en msec |         |      |      |         |        |      |         |       |       |
|------------------------|---------|------|------|---------|--------|------|---------|-------|-------|
| n                      | t1: sel | ins  | mer  | t2: sel | ins    | mer  | t3: sel | ins   | mer   |
| 100                    | 0.96    | 0.05 | 0.71 | 0.97    | 1.79   | 0.64 | 0.94    | 0.95  | 0.89  |
| 200                    | 3.40    | 0.10 | 1.52 | 3.46    | 6.93   | 1.37 | 3.36    | 3.44  | 1.97  |
| 300                    | 7.64    | 0.15 | 2.37 | 8.02    | 15.46  | 2.13 | 7.52    | 7.43  | 3.15  |
| 400                    | 13.41   | 0.20 | 3.26 | 13.80   | 27.59  | 3.02 | 13.37   | 14.08 | 4.38  |
| 500                    | 21.77   | 0.30 | 4.55 | 21.85   | 42.71  | 3.95 | 20.82   | 22.43 | 5.72  |
| 600                    | 30.39   | 0.30 | 5.21 | 31.13   | 62.16  | 4.64 | 30.34   | 32.18 | 7.19  |
| 700                    | 41.54   | 0.35 | 6.06 | 42.85   | 87.28  | 5.46 | 41.03   | 45.52 | 8.48  |
| 800                    | 53.88   | 0.40 | 7.10 | 56.05   | 118.52 | 6.33 | 54.20   | 57.36 | 9.92  |
| 900                    | 68.56   | 0.45 | 7.88 | 71.28   | 142.15 | 7.40 | 68.54   | 71.84 | 11.37 |
| 1000                   | 84.56   | 0.50 | 8.77 | 87.87   | 177.36 | 8.46 | 85.01   | 90.58 | 12.05 |

Interprétation des résultats :

- *liste croissante* : le tri par insertion est nettement plus rapide que les deux autres (à votre avis, pourquoi?) ; le tri par fusion est meilleur que le tri par sélection (l'écart augmente quand  $n$  augmente).
- *liste décroissante* : le tri par insertion est nettement plus lent que les deux autres ; le tri par fusion est meilleur que les deux autres (même remarque sur l'écart).
- *liste aléatoire* : les tris par insertion et sélection ont des temps du même ordre de grandeur, le tri par fusion est meilleur que les deux autres (même remarque sur l'écart).

Cet "écart" est intéressant : ce qui va nous intéresser est la *croissance* du temps d'exécution en fonction de  $n$ , plutôt que des chiffres. C'est ce qu'exprime la *complexité*.

### Efficacité des algorithmes du calcul de l'exposant

Les deux fonctions suivantes calculent  $a^n$  (cf. Chap. 8) :

```

1 def expo(a, n):
2 r = 1
3 for i in range(n):
4 r = a * r
5 return r

1 def expo(a, n):
2 r = 1
3 while n > 0:
4 if n % 2 == 0:
5 a = a * a
6 n = n // 2
7 else:
8 r = r * a
9 n = n - 1
10 return r

```

### Comparaison des algorithmes du calcul de l'exposant

Comparaison des deux fonctions pour le calcul de l'exposant :

```
Temps affichés en msec. Calcul de 2^n
n: class: amel:
1000 0.112 0.012
2000 0.266 0.014
3000 0.536 0.019
4000 0.776 0.024
5000 1.294 0.025
6000 1.654 0.031
7000 2.121 0.027
8000 3.036 0.037
9000 2.992 0.039
10000 3.636 0.051
```

Comment exprimer formellement que la deuxième version est plus efficace que la première ? Encore une fois, la notion de *complexité* va permettre d'y répondre.

## 9.4. La notation grand- $O$ et la notion de complexité dans le pire des cas

### Temps d'exécution

Pour analyser les temps d'exécution, nous allons définir une fonction  $T(n)$  qui va représenter le nombre d'unités de temps pris par un algorithme sur une entrée de taille  $n$ , et ce, *dans le pire des cas*.

Pour simplifier les calculs, on va considérer que les instructions élémentaires prennent chacune *1 unité de temps* : assignation, opération arithmétique sur des nombres, etc.

*Exemple* : soit la fonction suivante, comment exprimer  $T(n)$  où  $n$  est le nombre d'éléments de la liste donnée en entrée.

```
1 def sum_all(liste):
2 res = 0
3 for item in liste:
4 res += item
5 return res
```

$T(n) = 2n + 2$  car

- l'instruction 2 est exécutée 1 fois
- l'instruction 3 est exécutée  $n$  fois (assignation de chaque élément)
- l'instruction 4 est exécutée  $n$  fois
- l'instruction 5 est exécutée 1 fois

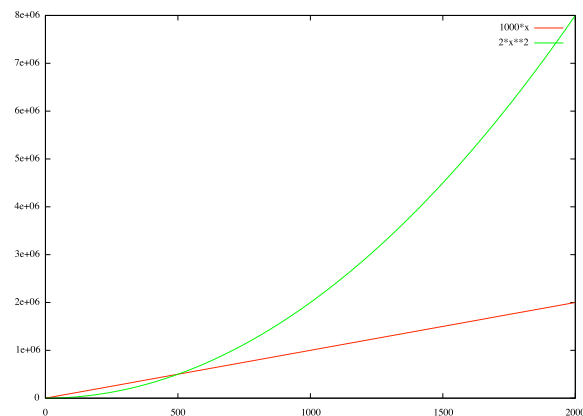
C'est une approximation (on n'a pas tenu compte de l'appel à la fonction, du temps exact de chaque instruction élémentaire, du nombre exact d'instructions élémentaires cachées dans l'instr. 3, etc.) mais cela a peu d'importance : ce qui va nous intéresser est la croissance de  $T(n)$ , pas les constantes.

Ce qui nous intéresse c'est la *croissance* de  $T(n)$ , pas les constantes.

Imaginons que chaque instruction élémentaire soit réalisée en  $1 \mu s$  (1 million d'instructions à la seconde) et que deux algorithmes  $A$  et  $B$  (pour le même problème) aient des temps  $T_A(n) = 1000n$  et  $T_B(n) = 2n^2$ .

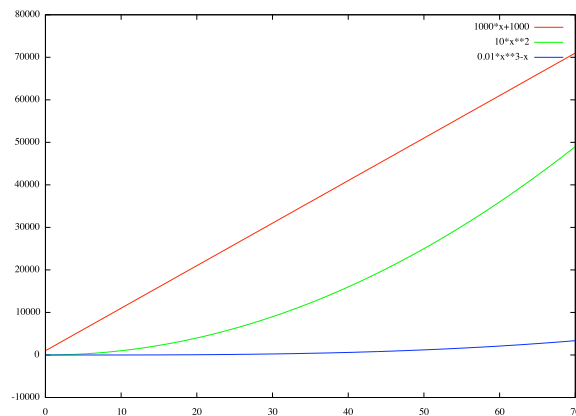
Temps d'exécution (en sec.) :

| $n$    | $A$   | $B$     |
|--------|-------|---------|
| 10     | 0.01  | 0.0002  |
| 100    | 0.1   | 0.02    |
| 1000   | 1.0   | 2.0     |
| 10000  | 10.0  | 200.0   |
| 100000 | 100.0 | 20000.0 |

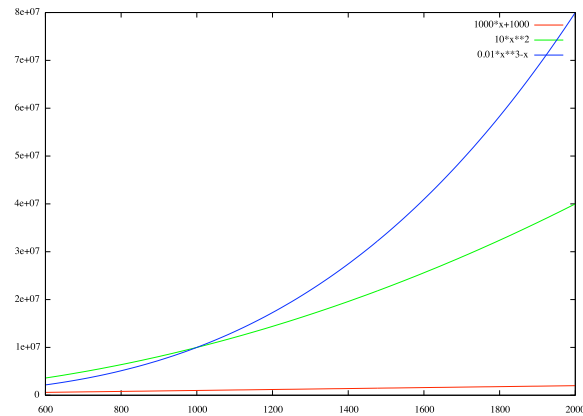


Dès que  $n > 500$ ,  $A$  est bien meilleur que  $B$  : la constante 1000 importe beaucoup moins que le fait que dans un cas on a une fonction linéaire, dans l'autre, elle est quadratique.

Comparaison de  $1000x + 1000$ ,  $10x^2$  et  $\frac{x^3}{100} - x$  sur  $[0, 70]$



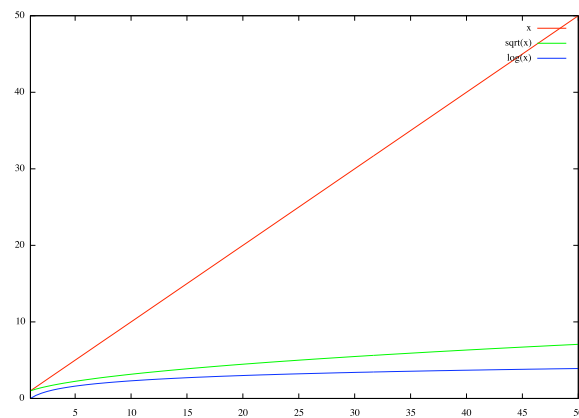
Comparaison de  $1000x + 1000$ ,  $10x^2$  et  $\frac{x^3}{100} - x$  sur  $[600, 2000]$



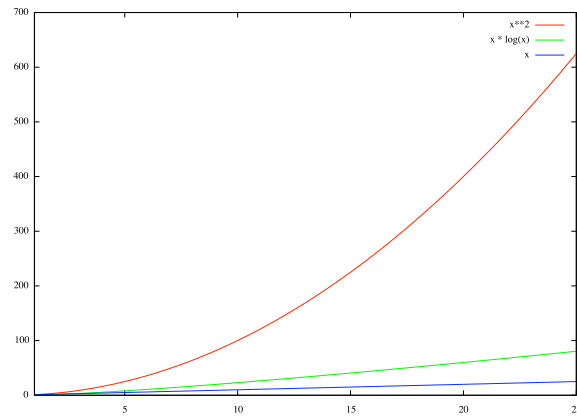
Les fonctions de temps suivantes sont classées par croissance. Les termes fréquemment utilisés pour décrire certains temps sont mis entre parenthèses.

- $T(n) = \log n$  (temps logarithmique, la base importe peu)
- $T(n) = \sqrt{n}$
- $T(n) = n$  (temps linéaire)
- $T(n) = n \log n$
- $T(n) = n^2$  (temps quadratique)
- $T(n) = n^3$  (temps cubique)
- $T(n) = 2^n$  (temps exponentiel, la base importe peu)
- $T(n) = n!$

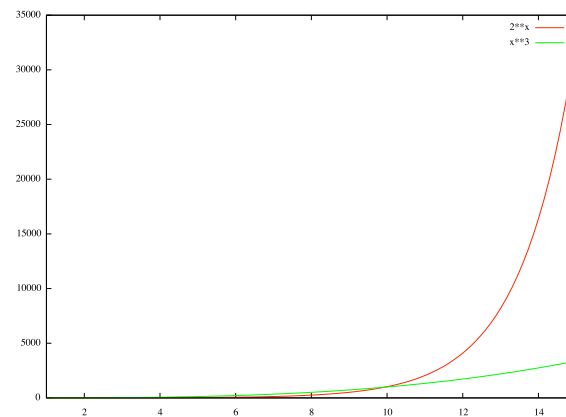
Comparaison de  $\log n$ ,  $\sqrt{n}$  et  $n$  sur  $[0, 50]$ .



Comparaison de  $n$ ,  $n \log n$  et  $n^2$  sur  $[0, 25]$ .



Comparaison de  $n^3$  et  $2^n$  sur  $[0, 15]$ .



Comparaison de  $2^n$  et  $n!$  sur  $[1, 20]$ .

```
>>> import math
>>> for i in range(1, 21):
>>> print('%3d %7d %19d' % (i, 2**i, math.factorial(i)))
```

|    |         |                     |
|----|---------|---------------------|
| 1  | 2       | 1                   |
| 2  | 4       | 2                   |
| 3  | 8       | 6                   |
| 4  | 16      | 24                  |
| 5  | 32      | 120                 |
| 6  | 64      | 720                 |
| 7  | 128     | 5040                |
| 8  | 256     | 40320               |
| 9  | 512     | 362880              |
| 10 | 1024    | 3628800             |
| 11 | 2048    | 39916800            |
| 12 | 4096    | 479001600           |
| 13 | 8192    | 6227020800          |
| 14 | 16384   | 87178291200         |
| 15 | 32768   | 1307674368000       |
| 16 | 65536   | 20922789888000      |
| 17 | 131072  | 355687428096000     |
| 18 | 262144  | 6402373705728000    |
| 19 | 524288  | 121645100408832000  |
| 20 | 1048576 | 2432902008176640000 |

### Taille maximale du problème, pour un temps donné

Imaginons qu'une machine exécute 1 million d'instructions à la seconde et que nous sommes prêts à attendre 1 heure de temps CPU pour résoudre un problème. Quelle est la taille maximale du problème que l'algorithme  $A$  ( $T_A(n) = 1000n$ ) peut résoudre ?

$$1000n \leq 60 \times 60 \times 10^6 = 36 \times 10^8,$$

donc  $n \leq 36 \times 10^5$ . L'algorithme A peut résoudre un problème de taille 3600000 en 1 heure.

Pour l'algorithme B ( $T_B(n) = 2n^2$ ) :

$$2n^2 \leq 36 \times 10^8,$$

donc  $n \leq \sqrt{18 \times 10^8} \simeq 42426$ .

Supposons qu'un algorithme dépense un temps  $f(n)$  pour résoudre un problème de taille  $n$  et que la machine exécute 1 million d'instructions à la seconde.

Taille maximale pour résoudre le problème en un temps donné :

| $f(n)$     | 1 sec.                 | 1 min.                   | 1 heure              | 1 jour                |
|------------|------------------------|--------------------------|----------------------|-----------------------|
| $\ln n$    | $3 \times 10^{434294}$ | $8 \times 10^{26057668}$ | $\simeq \infty$      | $\simeq \infty$       |
| $\sqrt{n}$ | $10^{12}$              | $3.6 \times 10^{15}$     | $1.3 \times 10^{19}$ | $7.5 \times 10^{21}$  |
| $n$        | $10^6$                 | $6 \times 10^7$          | $3.6 \times 10^9$    | $8.64 \times 10^{10}$ |
| $n \ln n$  | 87847                  | $3.9 \times 10^6$        | $1.8 \times 10^8$    | $3.9 \times 10^9$     |
| $n^2$      | 1000                   | 7745                     | 60000                | 293938                |

Supposons qu'un algorithme dépense un temps  $f(n)$  pour résoudre un problème de taille  $n$  et que la machine exécute 1 million d'instructions à la seconde.

Taille maximale pour résoudre le problème en un temps donné :

| $f(n)$ | 1 sec. | 1 min. | 1 heure | 1 jour | 1 mois | 1 an  | 1 siècle |
|--------|--------|--------|---------|--------|--------|-------|----------|
| $n^3$  | 100    | 391    | 1532    | 4420   | 13886  | 31595 | 146652   |
| $2^n$  | 13     | 16     | 31      | 36     | 41     | 44    | 51       |
| $n!$   | 9      | 11     | 12      | 13     | 15     | 16    | 17       |

### Notation grand-O

Soit  $T(n)$  un temps d'exécution mesuré en fonction de la taille  $n$  du problème. Nous pouvons supposer que  $T(n)$  est une fonction telle que :

- $n$  est un entier positif ou nul ;
- $T(n) \geq 0 \quad \forall n$ .

On va décrire le taux de croissance de ce type de fonction en utilisant une notation spécifique, la notation  $O$  (prononcer "grand-O"). On parle aussi de la *complexité* (sous-entendu "dans le pire des cas") de la fonction. Ceci permettra de classer les fonctions – et donc les temps d'exécution des algorithmes – selon leur type de croissance, indépendamment des constantes et autres détails techniques.

Les *fonctions de complexité* sont des fonctions

$$\mathbb{N} \rightarrow \mathbb{R}.$$

*Intuition* : Soit deux fonctions de complexité  $f(n)$  et  $g(n)$ , on note

$$f(n) \in O(g(n)),$$

ou

$$f(n) = O(g(n)),$$

si  $f$  ne croît pas plus vite que  $g$ .

On dit alors que “ $f$  est en grand- $O$  de  $g$ ”.

“ $f$  est en grand- $O$  de  $g$ ”

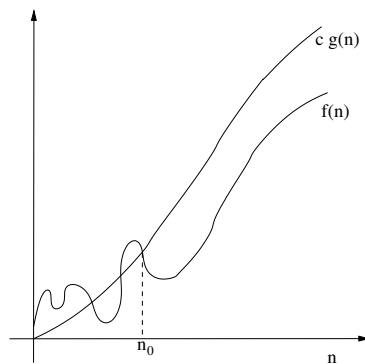
*Intuition* :  $f$  ne croît pas plus vite que  $g$ .

*Définition formelle* :

$$f(n) \in O(g(n)) \iff \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ \text{ t.q. } \forall n \geq n_0 : f(n) \leq c g(n).$$

*Preuves* : prouver qu’une fonction  $f(n) \in O(g(n))$  reviendra (principalement) à déterminer  $n_0$  et  $c$ .

$$f(n) \in O(g(n)) \iff \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ \text{ t.q. } \forall n \geq n_0 : f(n) \leq c g(n)$$



*Vision asymptotique* :

- $n_0 \geq 0$  permet d’éviter les effets de bord : ce n’est pas parce qu’un algorithme est plus rapide sur des petites valeurs, qu’il le sera encore asymptotiquement ;
- $c > 0$  exprime le fait que les constantes ne sont pas (très) importantes

$$f(n) \in O(g(n)) \iff \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ \text{ t.q. } \forall n \geq n_0 : f(n) \leq c g(n)$$

*Exemples* :

- Soit  $f(n) = 1000n$  et  $g(n) = n$ , alors  $f(n) \in O(g(n))$ . En effet, si  $n_0 = 0$  et  $c = 1000$ , alors

$$f(n) = 1000n \leq 1000n = 1000g(n), \quad \forall n \geq 0.$$

- Soit  $f(n) = 2n^2$  et  $g(n) = n^2$ , alors  $f(n) \in O(g(n))$ . En effet, si  $n_0 = 0$  et  $c = 2$ , alors  $f(n) \leq 2g(n)$ ,  $\forall n \geq 0$ .

$$f(n) \in O(g(n)) \iff \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ \text{ t.q. } \forall n \geq n_0 : f(n) \leq c g(n)$$

*Exemples* :

- Soit  $f(n) = (n+1)^2$  et  $g(n) = n^2$ , alors  $f(n) \in O(g(n))$ . En effet, si  $n_0 = 1$  et  $c = 4$ , alors

$$f(n) = (n+1)^2 = n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2 = 4g(n), \quad \forall n \geq 1.$$



Notez que l'inégalité n'est pas vraie si  $n = 0$ , d'où  $n_0 = 1$ .

Remarque : on aurait pu prendre d'autres valeurs (comme  $n_0 = 3$  et  $c = 2$ )

Exemples :

- $n^2 \in O(n^3)$ . Soit  $n_0 = 0$  et  $c = 1$  :  $n^2 \leq n^3, \forall n \geq 0$  est vrai (trivial).
- $n^3 \notin O(n^2)$ . Par l'absurde, supposons que  $n^3 \in O(n^2)$ , alors il existe deux constantes  $n_0$  et  $c$  telles que  $n^3 \leq c n^2, \forall n \geq n_0$ . Cela signifie que

$$c \geq \frac{n^3}{n^2} = n, \quad \forall n \geq \max(n_0, 1),$$

ce qui est une contradiction avec le fait que  $c$  est une constante (le  $\max(n_0, 1)$  est là pour éviter une division par zéro si  $n_0 = 0$ ).

### Simplification des expressions grand-O

La notation grand-O permet de simplifier les fonctions de complexité.

**Lemme** (les facteurs constants ne sont pas importants). *Pour toute valeur constante  $d > 0$  et toute fonction de complexité  $f(n)$ , on a*

$$f(n) \in O(d \cdot f(n)) \text{ et } d \cdot f(n) \in O(f(n))$$

*Preuve.* Soit  $c = \frac{1}{d}$  et  $n_0 = 0$ , on a

$$f(n) = (c \cdot d) f(n) = c(d \cdot f(n)), \quad \forall n \geq 0,$$

et donc  $f(n) \in O(d \cdot f(n))$ . Pour le deuxième cas, il suffit de choisir  $c = d$ . □

Exemples :  $1000n^2$  et  $\frac{n^2}{1000} \in O(n^2)$ .

**Lemme** (les termes d'ordre inférieurs sont négligeables). *Soit une fonction de complexité  $f(n)$  polynomiale et dont le coefficient du terme de plus grand degré est positif, c-à-d,*

$$f(n) := a_k n^k + a_{k-1} n^{k-1} + \dots + a_2 n^2 + a_1 n + a_0,$$

où  $a_k > 0$ . Alors,

$$f(n) \in O(n^k)$$

*Preuve.* Soit  $n_0 = 1$  et  $c = \sum_{i=0}^k \max(a_i, 0)$  (somme des coefficients positifs). Donc  $c$  est bien une constante strictement positive car  $a_k > 0$ . Alors,

$$f(n) \leq c n^k, \quad \forall n \geq 1.$$

En effet, pour chaque coefficient  $a_j$  :

- si  $a_j \leq 0$ , on a certainement  $a_j n^j \leq 0, \forall n \geq 1$ .
  - si  $a_j > 0$ , on a  $a_j n^j \leq a_j n^k, \forall n \geq 1$  car  $j \leq k$ .
- 

Exemple : Soit  $f(n) := 3n^5 + 10n^4 - 100n^3 + n + 1$ . Alors  $f(n) \in O(n^5)$ . En effet, si  $c = 3 + 10 + 1 + 1 = 15$ , alors  $f(n) \leq 15n^5, \forall n \geq 1$ .

On peut généraliser ce dernier lemme : si  $g(n)$  croît moins vite que  $h(n)$ , alors

$$g(n) + h(n) \in O(h(n)).$$

(on peut négliger ce qui croît moins vite)

*Exemple* : toute exponentielle croît plus vite que tout polynôme :

$$\lim_{n \rightarrow \infty} \frac{n^p}{a^n} = 0.$$

Donc, par exemple,

$$2^n + n^3 \in O(2^n).$$

**Lemme** (Transitivité des expressions grand- $O$ ). *Grand- $O$  est une relation transitive, c-à-d que si  $f \in O(g)$  et  $g \in O(h)$ , alors  $f \in O(h)$ .*

*Preuve.* Par définition,

$$f(n) \leq c_1 g(n), \quad \forall n \geq n_1,$$

$$g(n) \leq c_2 h(n), \quad \forall n \geq n_2.$$

Soit  $n_0 = \max(n_1, n_2)$  et  $c_0 = c_1 \cdot c_2$ , alors,

$$f(n) \leq c_1 g(n) \leq c_1 c_2 h(n) = c_0 h(n), \quad \forall n \geq n_0.$$

□

Pour exprimer la complexité d'une fonction  $T(n)$  qui représente le temps d'exécution d'un algorithme, nous suivrons les deux règles suivantes :

- *Simplification.* On exprime  $T(n)$  en appliquant les règles de simplification des expressions grand- $O$ . Ainsi, si  $T(n) = 2n^2 + 3$ , on dira que l'algorithme est exécuté en un temps  $O(n^2)$  (ou temps quadratique).
- *Proximité.* Il est clair que si  $T(n) = 2n^2 + 3$ , alors  $T(n) \in O(n^3)$  ou même  $O(2^n)$ , mais cela n'exprime pas la croissance de  $T(n)$ . On dira donc que  $T(n) \in O(n^2)$  (on choisit une fonction  $g$  dont la croissance est la plus faible possible et pour laquelle  $f \in O(g)$ .)

## 9.5. Evaluer la complexité des algorithmes itératifs

Nous allons appliquer la notation grand- $O$  pour évaluer la complexité des algorithmes, en suivant quelques règles qui permettent de simplifier l'analyse et qui découlent directement des simplifications des expressions grand- $O$ .

Au lieu d'évaluer un temps d'exécution  $T(n)$  précisément, les simplifications des expressions grand- $O$  vont être appliquées directement, en fonction des structures utilisées dans le programme à analyser (boucles, instructions conditionnelles, etc.)

- Une *instruction élémentaire* est une instruction qui réalise une action en un temps constant, quelle que soit la taille des entrées. Par exemple, les instructions suivantes sont des instructions élémentaires : affectation, opérations arithmétiques sur des entiers ou des réels, opérateur `[.]` d'accès à un élément d'une liste ou d'une chaîne, instruction `return`, etc.
- A l'inverse, toute instruction qui peut provoquer un temps d'exécution non constant n'est pas considérée comme une instruction élémentaire : appel d'une fonction ou d'une méthode (dont appels récurifs), boucles, instructions conditionnelles, certains opérateurs appliqués sur des objets de taille variable (comme l'opérateur "slice" sur les listes, son coût en temps peut ne pas être constant, dépendant de la taille de la tranche), etc.

*Règle (Instruction élémentaire).* Par définition, une instruction élémentaire à un coût en temps  $O(1)$  (temps constant).

### Somme et produit

On “additionne” la complexité de morceaux de code en ne gardant que la complexité la plus grande.

*Règle (Somme).* Soit deux parties  $A$  et  $B$  d’un programme, exécutées de façon séquentielle<sup>5</sup>. Supposons  $T_A(n) = O(f_A(n))$  et  $T_B(n) = O(f_B(n))$ . Alors

$$T_A(n) + T_B(n) = O(f_B(n) + f_A(n)) = O(\max(f_B(n), f_A(n))).$$

*Règle (Produit).* Soit deux fonctions  $f$  et  $g$ .

$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n)).$$

### Exemple

```
def swap(a, b):
 temp = b
 b = a
 a = temp
 return (a, b)
```

Le temps de cet algorithme est en  $O(1)$  car sa complexité est  $O(1) + O(1) + O(1) + O(1)$  (somme). De manière équivalente, il est en  $O(1)$  car  $4 \cdot O(1) = O(4) = O(1)$  (produit).

### Appel de fonction

*Règle (Appel de fonction ou méthode).* Le coût d’un appel à une fonction ou une méthode est équivalent au coût de l’exécution du corps de la fonction.

*Remarque :* on expliquera par après comment évaluer le coût d’une fonction récursive, mais la règle ci-dessus reste d’application.

### Exemples

```
(a, b) = swap(a, b)
(a, b) = swap(a, b)
```

Le temps de cet algorithme (totalement inutile) est  $O(1)$ .

```
def f(n):
 res = g(n)
 res += h(n)
 return res
```

Imaginons que le temps d’exécution de  $g(n)$  soit en  $O(n)$  et celui de  $h(n)$  soit en  $O(n^2)$ , alors le temps d’exécution de  $f(n)$  est en  $O(n) + O(n^2) + O(1) = O(n^2)$ .

### Complexité des opérations sur les listes

*Observation :* travailler (ajout / suppression) en *fin* de liste est plus efficace qu’au *début*

- obtenir la longueur de la liste ( $n = \text{len}(t)$ ) :  $O(1)$

---

5. l’une après l’autre, la partie  $B$  n’est pas incluse dans le travail de la partie  $A$  et vice-versa.

- accéder à un élément d'un indice donné ( $x = t[i]$ ) :  $O(1)$
- ajouter un élément en fin de liste ( $t.append(3)$ ) :  $O(1)$
- supprimer et récupérer le dernier élément ( $x = t.pop()$ ) :  $O(1)$
- insérer un élément  $x$  à un indice  $i$  donné :  $t.insert(i, x)$ 
  - pire des cas : au début de la liste ( $t.insert(0, 8)$ ) :  $O(n)$
  - meilleur des cas : en fin de liste ( $t.insert(len(t), 8)$ ) :  $O(1)$
- supprimer la première occurrence d'une valeur  $x$  ( $t.remove(x)$ ) :  $O(n)$

(d'après le site wiki python<sup>6</sup>)

- obtenir une tranche de taille  $k$  :  $O(k)$
- copier une liste :  $O(n)$  (par ex., la méthode `extend` copie la seconde liste à la fin de la première, donc  $O(k)$  si la seconde liste à  $k$  éléments)
- concaténer :  $O(n_1 + n_2)$
- multiplier (= copier) une liste  $k$  fois :  $O(kn)$
- rechercher un élément dans la séquence ( $x$  in  $s$ ) :  $O(n)$
- trouver le min. ou le max. d'une séquence ( $x = \min(t)$ ) :  $O(n)$
- trier une liste de taille  $n$  ( $t.sort()$ ) :  $O(n \log n)$
- l'appel à `range` est en  $O(1)$  (mais est souvent utilisé dans une boucle, qui aura typiquement  $n$  éléments).

### Exemples

```
t.append(3)
print(3 in t)
t2 = t * 2
```

Ce morceau de code a un temps linéaire :  $O(1) + O(n) + O(2n) = O(n)$

```
t2 = t * len(t)
```

Cette instruction a un temps quadratique :  $O(n \cdot n) = O(n^2)$

### Boucles for

Règle (Boucles `for`). Le temps d'exécution d'une boucle `for` est équivalent à

complexité de la création de la séquence + # d'itérations (dans le pire des cas)  $\times$   
complexité du corps de la boucle

Remarques :

- la notation grand- $O$  est utilisée dans ce chapitre pour calculer les temps d'exécution dans le pire des cas. Or, on peut sortir d'une boucle de différentes façons (`break`, `return`, condition d'arrêt), d'où le pire des cas sur le # d'itérations.
- $n \cdot O(1) = O(n)$  et pas  $O(1)$  : règle du produit
- pour multiplier une expression grand- $O$  par quelque chose qui dépend de la taille du problème :  $f(n) \cdot O(g(n)) = O(f(n) \cdot g(n))$  (règle du produit)

### Exemple

```
1 def sum_all(liste):
2 res = 0
3 for item in liste:
4 res += item
5 return res
```

Le temps de cet algorithme est  $O(n)$ , où  $n$  est le nombre d'éléments de la liste.

6. <http://wiki.python.org/moin/TimeComplexity>

*Preuve.* La liste est déjà créée (car passée en paramètre), il n'y a donc pas de coût lié à son utilisation. Les instructions 2 et 5 sont en  $O(1)$ . La boucle est exécutée  $n$  fois. Le temps d'exécution du corps de la boucle (instr. 4) est en  $O(1)$ . Donc le temps de cet algorithme est  $O(1) + n \cdot O(1) = O(n)$ .  $\square$

### Exemple

```
1 def sum_1_to_n(n):
2 res = 0
3 for i in range(1, n + 1):
4 res += i
5 return res
```

Le temps de cet algorithme est  $O(n)$ , où  $n$  est le nombre d'éléments de la liste.

*Preuve.* La création de la séquence (fonction `range`) est en  $O(1)$ . Les instructions 2 et 5 sont en  $O(1)$ . La boucle est exécutée  $n$  fois. Le temps d'exécution du corps de la boucle (instr. 4) est en  $O(1)$ . Donc le temps de cet algorithme est  $O(1) + O(n) + n \cdot O(1) = O(n)$ .  $\square$

### Boucles while

Règle (Boucles `while`). Le temps d'exécution d'une boucle `while` est équivalent à

# d'itérations (dans le pire des cas)  $\times$  (complexité de l'évaluation de la condition + complexité du corps de la boucle) + complexité pour sortir de la boucle (condition)

Remarques :

- à chaque itération, la condition est évaluée, il faut donc en tenir compte.
- pour sortir de la boucle, il faut encore évaluer une fois la condition.

### Exemple

```
1 def sum_1_to_n(n):
2 t = list(range(1, n + 1))
3 i = 0
4 res = 0
5 while i < n:
6 res += t[i]
7 i += 1
8 return res
```

Le temps de cet algorithme est  $O(n)$ , où  $n$  est le nombre d'éléments de la liste.

*Preuve.* Le temps de l'instr. 2 est en  $O(n)$ . Les instructions 3, 4 et 8 sont en  $O(1)$ . La boucle est exécutée  $n$  fois et son corps est en  $O(1)$ . Le temps de l'évaluation de la condition est en  $O(1)$ . Donc le temps de cet algorithme est  $O(n) + n \cdot (O(1) + O(1)) + O(1) = O(n)$ .  $\square$

### Exemple

```
1 def is_include(t1, t2):
2 i = 0
3 n1 = len(t1)
4 while i < len(t1) and t1[i] in t2:
5 i += 1
6 return i == n1
```

Cet algorithme retourne vrai ssi les éléments de  $t_1$  sont tous présents dans  $t_2$ . Le temps de cet algorithme est  $O(n_1 \cdot n_2)$ , où  $n_1$  et  $n_2$  sont les tailles de  $t_1$  et  $t_2$ .

*Preuve.* Le temps des instr. 2, 3 et 6 sont en  $O(1)$ . La boucle est exécutée  $n_1$  fois et son corps est en  $O(1)$ . Le temps de l'évaluation de la condition est en  $O(n_2)$ . Donc le temps de cet algorithme est  $O(1) + n_1 \cdot (O(1) + O(n_2)) + O(n_2) = O(n_1 \cdot n_2)$ .  $\square$

### Analyse du tri par insertion

Quelle est la complexité dans le pire des cas du tri par insertion, en fonction du nombre  $n$  d'éléments à trier ?

```

1 def insertion_sort(t):
2 n = len(t)
3 for i in range(1,n):
4 clef = t[i]
5 j = i - 1
6 while j >= 0 and t[j] > clef:
7 t[j+1] = t[j]
8 j = j - 1
9 t[j+1] = clef

```

La complexité du tri par insertion est en temps  $O(n^2)$ .

### Le tri par insertion est en $O(n^2)$ – preuve

- l'instruction 2 est en  $O(1)$
- considérons d'abord la boucle interne (`while`) : le corps de cette boucle est en  $O(1)$  (instr. 7 et 8) et, dans le pire des cas, on sort de la boucle en ayant décalé tous les éléments, c-à-d quand le test  $j \geq 0$  n'est plus vrai. Comme  $j$  est initialisé à  $i - 1$ , dans le pire des cas, on exécutera  $i$  fois la boucle (le nombre d'éléments entre l'indice 0 et l'indice  $i - 1$  est  $i$ ). Le temps de l'évaluation de la condition est en  $O(1)$ . La complexité totale de la boucle interne est donc  $O(i)$  (dans le pire des cas  $i = n - 1$  mais gardons cette expression en  $i$  pour le moment car il représente l'indice de la boucle extérieure)
- considérons maintenant la boucle externe (`for`) : elle est exécutée exactement  $n - 1$  fois. L'appel à `range` est en  $O(1)$ . Toutes les instructions du corps de la boucle sont en  $O(1)$ , sauf la boucle interne qui est en  $O(i)$  (voir ci-avant). On peut calculer le temps total de la boucle comme suit grâce à la formule d'Euler :

$$O(n) + \sum_{i=0}^{n-1} O(i) = O(0 + 1 + \dots + n - 1 + n) = O\left(\sum_{i=1}^n i\right) = O\left(\frac{n^2 + n}{2}\right)$$

- On en conclut que le temps du tri par insertion est en  $O(n^2)$   $\square$

### Tri par insertion : et si la liste est déjà triée ?

- si la liste est triée de manière croissante, alors l'analyse de la boucle intérieure montre qu'elle sera exécutée en temps constant (le test  $t[j] > clef$  étant faux pour chaque  $i$ )
- l'analyse complète donne alors un temps en  $O(n)$  dans ce cas particulier, qui est en réalité le *meilleur* des cas
- cette analyse de la complexité explique les temps CPU observés

### Instructions conditionnelles

Dans le cas d'une instruction conditionnelle, on ne garde que le plus grand temps d'exé-

cution de toutes les *branches*<sup>7</sup>.

**Règle (Instructions conditionnelles).** Soit une instruction conditionnelle avec  $k$  branches et  $\ell$  conditions à évaluer, et soit  $T_i(n)$  le temps d'exécution de la  $i$ ème branche. Si  $T_i(n) \in O(f_i(n))$  pour  $i = 1, \dots, k$ , et si l'évaluation de la condition  $j$  est en  $O(g_j(n))$ , pour  $j = 1, \dots, \ell$  alors

$$T(n) \in O\left(\max_i(f_i(n)) + \max_j(g_j(n))\right),$$

où  $T(n)$  est le temps d'exécution de l'instruction conditionnelle complète.

**Exemple :** attention aux opérations et appels de fonctions / méthodes cachés dans les conditions

```
if x in t:
 print 'x est dans t'
else:
 print 'x pas dans t'
```

- les deux branches sont en  $O(1)$
- le test du `if` est en  $O(n)$  (où  $n = \text{len}(t)$ )
- la complexité est donc  $O(n) + \max(O(1), O(1)) = O(n)$

### Analyse du tri par sélection

Quelle est la complexité dans le pire des cas du tri par sélection, en fonction du nombre  $n$  d'éléments à trier ?

```
1 def selection_sort(t):
2 n = len(t)
3 for i in range(n-1):
4 small = i
5 for j in range(i+1, n):
6 if t[j] < t[small]:
7 small = j
8 (t[i], t[small]) = (t[small], t[i])
```

La complexité du tri par sélection est en temps  $O(n^2)$ .

### Le tri par sélection est en $O(n^2)$ – preuve

- l'instruction 2 est en temps constant ( $O(1)$ ).
- la boucle interne (instr. 5 à 7) est en  $O(n-i)$  :
  - le corps de la boucle est en  $O(1)$  car la condition du `if` et l'instr. 7 sont en  $O(1)$
  - la construction de la séquence est en  $O(n-i-1)$ .
  - la boucle est exécutée exactement  $n-i-1$  fois.
  - le temps total de cette boucle est donc  $O(n-i-1) + O(n-i-1) \times O(1) = O(n-i)$
- la boucle externe (instr. 3 à 8) est en  $O(n^2)$  :
  - la construction de la séquence est en  $O(n-1)$ .
  - la boucle est exécutée exactement  $n-1$  fois.
  - le corps de la boucle est en  $O(n-i)$  (coût de la boucle interne) car les instructions 4 et 8 sont en temps constant.
  - le temps total de cette boucle est donc

$$O(n-1) + \sum_{i=0}^{n-2} O(n-i) = O(n-1) + O(n + (n-1) + \dots + 2) = O(n^2),$$

7. Voir chap. 4.

par la formule d'Euler (cf. preuve du tri par insertion)  $\square$

### Tri par sélection : et si la liste est déjà triée ?

```

1 def selection_sort(t):
2 n = len(t)
3 for i in range(n-1):
4 small = i
5 for j in range(i+1, n):
6 if t[j] < t[small]:
7 small = j
8 (t[i], t[small]) = (t[small], t[i])

```

Le temps du tri par sélection est insensible au fait que la liste soit triée ou non et reste donc en  $O(n^2)$  (le nombre d'itérations des deux boucles sera identiques, quelque soit la manière dont les données sont ordonnées).

## 9.6. Evaluer la complexité des algorithmes récursifs

### Evaluer la complexité des algorithmes récursifs

Pour analyser le temps d'exécution  $T(n)$  d'une fonction *récursive*  $f$  :

- on détermine une formule *inductive* pour  $T(n)$  :
  - *Base*. Que vaut  $T(n)$  quand il n'y a pas d'appel récursif ?
  - *Gén.* Que vaut  $T(n)$  quand il y a des appels récursifs ? On estime le temps  $T(n)$  par rapport au temps  $T(k)$  des appels récursifs où  $k$  est la taille appropriée des arguments (par exemple,  $T(n-1)$ ).
- on évalue *deux fois* le temps du corps de  $f$  comme précédemment, mais en gardant les termes  $T(k)$  sous la forme d'inconnues.
  - quand on tombe dans le(s) cas de base
  - dans le cas général
- Dans les expressions obtenues, on remplace les termes grand- $O$  comme  $O(h(n))$  par  $c \cdot h(n)$  où  $c$  est une constante particulière.
- On en déduit une forme *non-inductive* de  $T(n)$

*Exemple*

```

1 def sum_rec(n):
2 if n == 1:
3 return 1
4 else:
5 return sum_rec(n-1) + n

```

- pour la base de la définition inductive de  $T(n)$ , on choisit  $n = 1$ . Dans ce cas, la complexité de  $T(1)$  est en  $O(1)$  (instr. 2 et 3).
- si  $n > 1$ , seules les instr. 2 et 4 et 5 sont exécutées. Les instructions 2 et 4 sont en  $O(1)$  et le temps d'exécution de l'instr. 5 est  $T(n-1) + O(1)$ .
- la forme *inductive* de  $T(n)$  est donc :

$$\begin{aligned}
 T(1) &= O(1), \\
 T(n) &= T(n-1) + O(1), \quad \forall n > 1.
 \end{aligned}$$

- on remplace les expressions  $O(h(n))$  par  $c \cdot h(n)$  (une constante pour chaque ex-



pression) :

$$\begin{aligned} T(1) &= a, \\ T(n) &= T(n-1) + b, \quad \forall n > 1. \end{aligned}$$

- on exprime  $T(n)$  en fonction de  $n$  et des constantes. Pour les premières valeurs on a :

$$\begin{aligned} T(1) &= a, \\ T(2) &= T(1) + b = a + b, \\ T(3) &= T(2) + b = a + 2b, \\ T(4) &= T(3) + b = a + 3b. \end{aligned}$$

- on déduit que  $T(n) = a + (n-1)b$ ,  $\forall n \geq 1$  (forme *non-inductive*)
- on conclut que  $T(n) \in O(n)$  puisque  $a$  et  $b$  sont des constantes

*Remarque* : nous nous sommes basés sur des "expérimentations" pour déterminer la forme non-inductive

$$T(n) = a + (n-1)b, \quad \forall n \geq 1.$$

Ce n'est donc pas une preuve. Mais il est simple de prouver par récurrence (exercice) que cette formule implique la forme inductive

$$\begin{aligned} T(1) &= a, \\ T(n) &= T(n-1) + b, \quad \forall n > 1. \end{aligned}$$

### Analyse du tri par fusion

Quelle est la complexité dans le pire des cas du tri par fusion, en fonction du nombre  $n$  d'éléments à trier ?

```

1 def merge_sort(t):
2 n = len(t)
3 if n > 1:
4 (t1, t2) = split(t)
5 t1 = merge_sort(t1)
6 t2 = merge_sort(t2)
7 return merge(t1, t2)
8 else:
9 return t
10
11 def split(t):
12 """ precondition: len(t) >= 2 """
13 mid = len(t) // 2
14 t1 = t[:mid]
15 t2 = t[mid:]
16 return (t1, t2)
17
18 def merge(t1, t2):
19 if len(t1) == 0:
20 return t2
21 elif len(t2) == 0:
22 return t1
23 elif t1[-1] > t2[-1]:
24 last = t1.pop()
25 new = merge(t1, t2)
26 new.append(last)
27 return new
28 else:
29 last = t2.pop()
30 new = merge(t1, t2)
31 new.append(last)
32 return new

```

La complexité du tri par fusion est en temps  $O(n \log_2 n)$ , ce qui est bien meilleur que les tris par insertion et par sélection !

**Le tri par fusion est en  $O(n \log_2 n)$  – preuve**

Nous allons analyser les 3 fonctions utilisées dans le tri par fusion et montrer que :

- `split` est en  $O(n)$
- `merge` est en  $O(n)$
- `merge_sort` est en  $O(n \log_2 n)$

*Notations* : dans ce qui suit,  $n$  représente la taille de la liste  $t$ ,  $n_1$  la taille de  $t_1$  et  $n_2$  celle de  $t_2$ .

$\implies$  notez que  $n = n_1 + n_2$

```

11 def split(t):
12 """ precondition: len(t) >= 2 """
13 mid = len(t) // 2
14 t1 = t[:mid]
15 t2 = t[mid:]
16 return (t1, t2)

```

*Analyse de la fonction `split`*

- L'opérateur "slice" a une complexité  $O(k)$  où  $k$  est la taille de la tranche retournée
- L'instruction 14 est donc en  $O(n_1)$  et l'instruction 15 est en  $O(n_2)$
- Les autres instructions sont en temps constant
- Le temps total de la fonction `split` est donc  $O(n_1 + n_2) = O(n)$   $\square$

```

18 def merge(t1, t2):
19 if len(t1) == 0:
20 return t2
21 elif len(t2) == 0:
22 return t1
23 elif t1[-1] > t2[-1]:
24 last = t1.pop()
25 new = merge(t1, t2)
26 new.append(last)
27 return new
28 else:
29 last = t2.pop()
30 new = merge(t1, t2)
31 new.append(last)
32 return new

```

*Analyse de la fonction `merge`*

- Si  $n_1 = 0$ , on retourne (une référence vers) la liste  $t_2$ , ce qui se fait en temps constant  $O(1)$ . Le cas  $n_2 = 0$  est symétrique.
- Donc, si une des listes  $t_1$  ou  $t_2$  est vide  $\Rightarrow O(1)$
- Evaluons  $T(n)$  pour `merge` où  $n = n_1 + n_2$ 
  - Base. Si  $n = 1$ , alors une des deux listes ( $t_1$  ou  $t_2$ ) est vide et donc  $T(1) = O(1)$
  - Base.  $T(1) = O(1)$
  - Gén. Si  $n > 1$  et qu'une des deux listes est vide, alors  $T(n) = O(1)$ . Si aucune des deux listes n'est vide, considérons le premier cas ( $t_1[-1] > t_2[-1]$ ) (le second cas est symétrique). La condition est en  $O(1)$  (instr. 23). Le coût du corps de cette branche (instr. 24 à 27), hors appel récursif, est en  $O(1)$  car toutes les instructions sont en temps constant (travail en fin de liste). Lors de l'appel récursif,  $t_1$  a un élément de moins. Donc,  $T(n) = O(1) + T(n-1)$ .
  - Gén. Si  $n > 1$ ,  $T(n) = O(1) + T(n-1)$
  - La forme inductive de  $T(n)$  est donc *identique* à celle vue pour `sum_rec` ci-avant et on peut en conclure que `merge` est aussi en  $O(n)$   $\square$

```

1 def merge_sort(t):
2 n = len(t)
3 if n > 1:
4 (t1, t2) = split(t)
5 t1 = merge_sort(t1)
6 t2 = merge_sort(t2)
7 return merge(t1, t2)
8 else:
9 return t

```

*Analyse de la fonction `merge_sort`*

- Base. Si  $n = 1$ , alors  $T(1) = O(1)$ .

- Gén. Si  $n > 1$ , l'instr. 4 est en  $O(n)$  (appel à `split`) et l'instr. 7 également (appel à `merge`). Les appels récursifs se font sur deux moitiés de  $t$ . Pour simplifier, on suppose que  $n$  est une puissance de 2. On obtient donc :

$$T(n) = O(n) + 2 T\left(\frac{n}{2}\right),$$

où  $n$  est une puissance de 2 ( $> 1$ )

*Remarques :*

- Supposer que  $n$  est une puissance de 2 est pratique car les listes seront toujours séparées en deux parties égales ( $\frac{n}{2}$ ), jusqu'à tomber dans un cas de base (liste de longueur 1)
- Comme  $T(n)$  ne décroît sûrement pas quand  $n$  croît, le fait d'étudier la croissance de  $T(n)$  pour les puissances de 2 est suffisant. En effet, le fait de sauter certaines valeurs de  $T(n)$  ne change pas sa croissance.

Avec les constantes, cela donne :

$$\begin{aligned} T(1) &= a, \\ T(n) &= bn + 2 T\left(\frac{n}{2}\right), \quad n > 1. \end{aligned}$$

Les premières valeurs sont :

$$\begin{aligned} T(1) &= a, \\ T(2) &= 2b + 2 T(1) = 2a + 2b, \\ T(4) &= 4b + 2 T(2) = 4a + 8b, \\ T(8) &= 8b + 2 T(4) = 8a + 24b, \\ T(16) &= 16b + 2 T(8) = 16a + 64b. \end{aligned}$$

Le coefficient de  $a$  semble être  $n$ . Analysons les coefficients de  $b$  :

| $n$                         | 2 | 4 | 8  | 16 |
|-----------------------------|---|---|----|----|
| Coef. de b                  | 2 | 8 | 24 | 64 |
| Ratio Coef./ $n = \log_2 n$ | 1 | 2 | 3  | 4  |

Il apparaît que le ratio est  $\log_2 n$  et donc que le coefficient de  $b$  est  $n \log_2 n$ . D'une manière générale, on peut prouver par récurrence (exercice) que si  $n$  est une puissance de 2 telle que

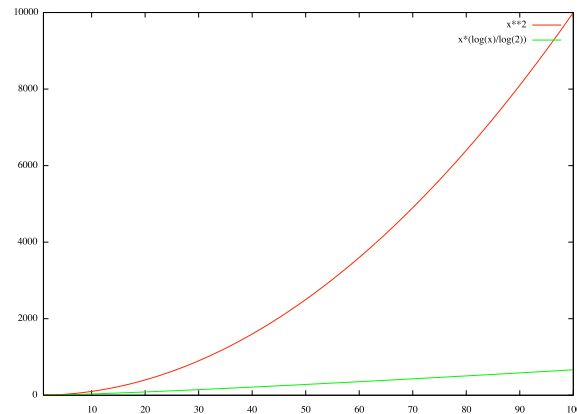
$$\begin{aligned} T(1) &= a, \\ T(n) &= bn + 2 T\left(\frac{n}{2}\right), \quad n > 1, \end{aligned}$$

alors  $T(n) = an + bn \log_2 n$ . On en conclut que  $T(n) \in O(n \log_2 n)$   $\square$

## 9.7. Analyse du comportement de nos algorithmes

### Comparaison de la complexité des tris

Le tri par fusion est en  $O(n \log_2 n)$  et les deux autres algorithmes de tris sont en  $O(n^2)$ .



Il existe beaucoup d'autres algorithmes de tris, joliment illustrés sur le site : <https://www.toptal.com/developers/sorting-algorithms>.

La meilleure complexité possible pour un tri en  $O(n \log n)$  (par ex., tri par fusion et quicksort).

### Comparaison de la complexité du calcul de l'exposant

```

1 def expo(a, n):
2 r = 1
3 for i in range(n):
4 r = a * r
5 return r

1 def expo(a, n):
2 r = 1
3 while n > 0:
4 if n % 2 == 0:
5 a = a * a
6 n = n // 2
7 else:
8 r = r * a
9 n = n - 1
10 return r

```

La version simple calcule  $a^n$  en  $O(n)$  (preuve laissée en exercice) et la version améliorée en  $O(\log_2 n)$ .

### La version améliorée du calcul de l'exposant est en $O(\log_2 n)$ – preuve

```

1 def expo(a, n):
2 r = 1
3 while n > 0:
4 if n % 2 == 0:
5 a = a * a
6 n = n // 2
7 else:
8 r = r * a
9 n = n - 1
10 return r

```

- il est clair que le cas pair diminue  $n$  plus vite
- si  $n$  était une puissance de 2, c'est à dire  $n = 2^p$  où  $p \geq 1$ 
  - on tomberait toujours dans le cas pair jusqu'à obtenir  $n = 1$
  - il y aurait exactement  $p + 1$  itérations, où  $p = \log_2 n$  (meilleur des cas)
- Imaginons maintenant le pire des cas :  $n$  est impair, et chaque fois que l'on divise  $n$  par 2, on retombe dans le cas impair
  - dans ce cas, le nombre d'itérations peut être doublé par rapport au meilleur cas

— on conclut que le temps d'exécution dans le pire des cas est  $O(2\log_2 n) = O(\log_2 n)$   $\square$

### Comparaison de la complexité du calcul de Fibonacci

```

1 def fib_rec(n):
2 if n == 0:
3 return 0
4 elif n == 1:
5 return 1
6 else:
7 return fib_rec(n-1) + fib_rec(n-2)

1 def fib_iter(n):
2 f = [0, 1]
3 for i in range(2, n+1):
4 f.append(f[i-1] + f[i-2])
5 return f[n]
```

La version récursive calcule  $f_n$  en  $O(\varphi^n)$  et la version itérative en  $O(n)$  ( $\varphi = \frac{1+\sqrt{5}}{2} \simeq 1.618$  est le nombre d'or).

### La version itérative de Fibonacci est en $O(n)$ – preuve

```

1 def fib_iter(n):
2 f = [0, 1]
3 for i in range(2, n+1):
4 f.append(f[i-1] + f[i-2])
5 return f[n]
```

- les instructions 2 et 5 sont en  $O(1)$
- grâce à `append`, l'instruction 4 (et donc le corps de la boucle) est en  $O(1)$
- la séquence créée par `range` et le nombre d'itérations sont en  $O(n)$
- au total :  $O(1) + O(n) + O(n) \times O(1) = O(n)$   $\square$

### La version récursive de Fibonacci est en $O(1.618^n)$ – preuve intuitive

```

1 def fib_rec(n):
2 if n == 0:
3 return 0
4 elif n == 1:
5 return 1
6 else:
7 return fib_rec(n-1) + fib_rec(n-2)
```

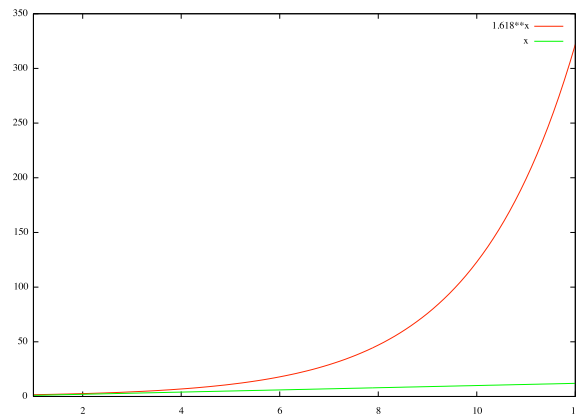
Pas une preuve formelle mais explique intuitivement le résultat :

- Clairement  $T(n) = T(n-1) + T(n-2)$
- on peut penser que chaque appel récursif "double" approximativement le nombre d'appels
  - On aurait alors une fonction du type  $T(n) = 2^n$
  - Mais cela ne marche pas car  $2^n \neq 2^{n-1} + 2^{n-2}$
- Gardons l'idée d'une exponentielle  $T(n) = a^n$  et essayons de déterminer sa base.
- Comme  $T(n) = T(n-1) + T(n-2)$ , il s'agit de résoudre

$$a^n = a^{n-1} + a^{n-2}$$

- En divisant les deux membres par  $a^{n-2}$  on obtient  $a^2 = a + 1$
- Les racines de cette équation sont  $\frac{1+\sqrt{5}}{2} \simeq 1.618$  et  $\frac{1-\sqrt{5}}{2} \simeq -0.618$
- Comme les fonctions de complexité doivent retourner quelque chose de positif (un temps), on ne garde que la première solution, qui est le nombre d'or.  $\square$

La complexité du calcul récursif de Fibonacci est donc exponentielle !



## 9.8. Glossaire

*temps CPU* : temps mis par le processeur pour exécuter un programme, le temps CPU varie d'une machine à l'autre et est difficile à estimer précisément.

*instruction élémentaire* : instruction qui est exécutée en temps constant.

*notation grand-O* : notation qui permet d'exprimer la croissance d'une fonction, de manière simplifiée.

*complexité* : la complexité d'un algorithme est la croissance, dans le pire des cas, de son temps d'exécution. Elle est exprimée grâce à la notation grand-O en fonction de la taille des entrées.

## Dictionnaires

*Dictionnaires • Exemple d'utilisation des dictionnaires • Complexité des opérations sur les dictionnaires • Quelques algorithmes sur des dictionnaires • Glossaire*

---

### 10.1. Dictionnaires

Un *dictionnaire* est une séquence mutable, comme une liste, mais est plus générale :

- dans une liste, les indices doivent être des entiers ;
- dans un dictionnaire, ils peuvent être de (presque) n'importe quel type.

Un dictionnaire peut être vu comme une correspondance entre un ensemble d'indices (les *clefs*) et de valeurs.

- à chaque clef correspond une valeur ;
- l'association entre une clef et une valeur est une *paire clef-valeur* ou un *élément* de la séquence.

*Syntaxe.*

- lors de sa création, on utilise des accolades `{ . }`. Pour rappel, les listes utilisent des crochets `[ . ]` et les tuples des parenthèses `( . )`.
- pour associer des clefs aux valeurs, on écrit une séquence `clef : valeur` séparés par des virgules.
- on peut également utiliser l'opérateur d'accès pour modifier un dictionnaire `d[clef] = valeur`.

*Exemple.*

Construisons un dictionnaire anglais-français : les clefs et les valeurs seront des chaînes de caractères.

```
>>> eng2fr = {}
>>> type(eng2fr)
<class 'dict'>
>>> eng2fr = {'one': 'un', 'two' : 'deux', 'three' : 'trois'}
>>> eng2fr['four'] = 'quatre'
>>> print(eng2fr)
{'four': 'quatre', 'three': 'trois', 'two': 'deux', 'one': 'un'}
```

L'ordre des valeurs peut paraître surprenant.

```
>>> print(eng2fr)
{'four': 'quatre', 'three': 'trois', 'two': 'deux', 'one': 'un'}
```

En réalité, cet ordre peut même dépendre de la machine sur laquelle vous exécutez ces instructions. 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 clefs.

```
>>> print(eng2fr['two'])
deux
>>> print(eng2fr['five'])
KeyError: 'five'
```

Concernant les types :

- les valeurs peuvent être de n'importe quel type (comme pour les listes)
- les clefs ne peuvent pas être d'un type mutable. Ex. de types acceptés : chaînes de caractères, entiers, ou tuples (cf. chap. suivant).
- Plus précisément, le type d'une clef doit être "hachable", c-à-d que l'on peut le convertir en un nombre entier d'une manière déterministe. Les objets mutables ne possèdent pas cette propriété.

```
>>> hash(6)
6
>>> hash('hello')
731779994702800104
>>> hash((1, 2))
3713081631934410656
>>> hash([1, 2, 3])
TypeError: unhashable type: 'list'
```

*Exemple :*

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

TypeError: unhashable type: 'list'
>>> d['dico']['two']
'deux'
```

## Un dictionnaire est une séquence

Un dictionnaire est une séquence : on peut lui appliquer certaines opérations déjà vues avec les chaînes et les listes.

- fonction `len`

```
>>> len(eng2fr)
4
```

- opérateur `in` : il s'applique pour rechercher une clef, pas une valeur (*intuition* : nous avons construit un dictionnaire anglais-français, pas l'inverse).



```
>>> 'one' in eng2fr
True
>>> 'deux' in eng2fr
False
```

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

On peut appliquer une boucle `for` sur un dictionnaire : elle traverse les *clefs* du dictionnaire.

```
>>> for k in eng2fr:
 print(k, eng2fr[k])

three trois
two deux
four quatre
one un
```

Pour itérer sur les valeurs, on peut utiliser la méthode `values` qui retourne un objet itérable `dict_values`.

```
>>> vals = eng2fr.values()
>>> type(vals)
<class 'dict_values'>
>>> print(vals)
dict_values(['trois', 'deux', 'quatre', 'un'])
>>> for v in vals:
 print(v)

trois
deux
quatre
un
>>> valeurs = list(vals)
>>> print(valeurs)
['trois', 'deux', 'quatre', 'un']
```

Si on veut obtenir les clefs d’un dictionnaire de façon triée, on peut utiliser la fonction `sorted` qui retourne une liste triée (cette fonction s’applique en réalité sur tous les objets itérables dont les éléments sont comparables).

```
>>> sorted(eng2fr)
['four', 'one', 'three', 'two']
>>> for k in sorted(eng2fr):
 print(k, ': ', eng2fr[k])

four : quatre
one : un
three : trois
two : deux
```

## Supprimer un élément d’un dictionnaire

La méthode `pop` existe aussi pour les dictionnaires.

```
>>> print(eng2fr)
{'four': 'quatre', 'one': 'un', 'two': 'deux', 'three': 'trois'}
>>> removed_element = eng2fr.pop('one')
>>> print(eng2fr)
{'four': 'quatre', 'two': 'deux', 'three': 'trois'}
>>> print(removed_element)
un
```

## 10.2. Exemple d'utilisation des dictionnaires

### Compter la fréquence des lettres

*Problème.* Comment compter la fréquence de chaque lettre dans une chaîne de caractères ?

Plusieurs solutions :

- utiliser 26 variables (compteurs), une pour chaque lettre.
- utiliser une liste de 26 compteurs.
- utiliser un dictionnaire.

Essayons ces possibilités.

*Solution 1 :* 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!).

*Démonstration du code en séance :* il est effrayant.

*Solution 2 :* créer une liste de 26 éléments puis convertir chaque caractère en un nombre (en utilisant la fonction `ord`). Utiliser ce nombre comme indice de la liste pour incrémenter le compteur approprié. Pour réafficher une lettre à partir d'un indice, utiliser `enumerate` et `chr`.

*Démonstration du code en séance :* c'est déjà mieux mais le travail sur les indices est gênant.

*Solution 3 :* créer un dictionnaire avec les caractères comme clefs 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.

*Démonstration du code en séance :* code simplifié et qui n'utilise que les compteurs réellement utiles. De plus, il associe directement une lettre (la clef) à ce qu'on veut calculer sur celle-ci.

## 10.3. Complexité des opérations sur les dictionnaires

Les dictionnaires ont une structure (sous-jacente) en Python qui leur permet des performances très intéressantes pour accéder à un élément *en moyenne* (mais pas toujours dans le pire des cas). Ils utilisent ce qu'on appelle une table de hachage<sup>1</sup>.

En pratique, la complexité en moyenne sera souvent atteinte.

|                                                    | Moyenne | Pire des cas |
|----------------------------------------------------|---------|--------------|
| Accès à un élément (via la clef)                   | $O(1)$  | $O(n)$       |
| Mettre à jour un élément (via la clef)             | $O(1)$  | $O(n)$       |
| Supprimer un élément (via la clef)                 | $O(1)$  | $O(n)$       |
| Obtenir un objet itérable (de clefs ou de valeurs) | $O(n)$  | $O(n)$       |

Le nombre des paires dans le dictionnaire est  $n$ . Source : <http://wiki.python.org/moin/TimeComplexity>

Pour tester cette complexité en moyenne, nous allons réaliser le test suivant :

- lire le fichier `words.txt` pour créer une liste  $t$  et un dictionnaire  $d$  contenant les 113809 mots;

1. Voir cours *Structure des données I et II* en BAC 2 et 3.

- utiliser le module `umons_cpu` (cf. Chap. 9) pour évaluer le temps d'exécution de la recherche d'un mot :
  - avec une recherche linéaire dans la liste (complexités moyenne et pire des cas en  $O(n)$ );
  - avec une recherche dichotomique dans la liste, puisque les mots sont triés (complexités en moyenne et pire des cas en  $O(\log_2 n)$ );
  - avec accès direct dans un dictionnaire (complexité en moyenne en  $O(1)$ , dans le pire des cas en  $O(n)$ ).

Création des données :

```
def get_data_structures():
 t = []
 d = {}
 with open('words.txt') as file:
 for line in file:
 word = line.strip()
 t.append(word)
 d[word] = ''
 return t, d
```

Recherche linéaire :

```
def linear_search(t, w):
 for word in t:
 if word == w:
 return True
 return False
```

Recherche dichotomique (vue lors des travaux pratiques) :

```
def dichotomic_search(t, w):
 start = 0
 end = len(t) - 1
 mid = (end - start) // 2
 while (end - start > 1) and w != t[mid]:
 if w < t[mid]:
 end = mid - 1
 else:
 start = mid + 1
 mid = start + (end - start) // 2
 return len(t) > 0 and w == t[mid]
```

Recherche dans le dictionnaire :

```
def dico_search(d, w):
 return w in d
```

Pour obtenir un mot aléatoire :

```
import random

def random_word(t):
 index = random.randint(0, len(t)-1)
 return t[index]
```

Résultats obtenus en faisant appel au module `umons_cpu` : on fait deux types de tests (mots existants ou non).

```
Temps affichés en msec
Tests sur mots aleatoires (existants):
linear dico dict
1.346363 0.007371 0.000176
Tests sur mots non presents:
linear dico dict
4.883696 0.006991 0.000177
```

*Remarque* : dans les tests sur les mots aléatoires, le coût de la création d'un mot aléatoire est inclus dans les temps ci-dessus, mais il est le même pour les 3 méthodes.

## 10.4. Quelques algorithmes sur des dictionnaires

Trouver une valeur à partir d'une clef est trivial, mais le contraire ?

*Problème*. Comment trouver la (les) clef(s) d'une valeur donnée ?

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

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

>>> months = {'Jan' : 31, 'Feb' : 28, 'Mar' : 31, 'Apr' : 30, 'May' : 31,
'Jun' : 30, 'Jul' : 31, 'Aug' : 31, 'Sep' : 30, 'Oct' : 31, 'Nov' : 30,
'Dec' : 31}

>>> print('Months with 31 days:', get_keys(months, 31))
Months with 31 days: ['Jan', 'Jul', 'Mar', 'Aug', 'Dec', 'May', 'Oct']
```

**Complexité** : pour rappel, `append` est en  $O(1)$ . On a donc un temps en  $O(n)$  en moyenne et en  $O(n^2)$  dans le pire des cas (dû à `d[k]`).

*Retour sur Fibonacci* : les dictionnaires permettent d'écrire le calcul de Fibonacci récursivement, tout en utilisant le principe de la version itérative : pour éviter de recalculer un grand nombre de fois les mêmes valeurs, on les stocke en mémoire.

```
known = {0 : 0, 1 : 1}

def fib_dict(n, known):
 if n in known:
 return known[n]
 res = fib_dict(n-1, known) + fib_dict(n-2, known)
 known[n] = res
 return res
```

Comparaison des temps CPU des 3 méthodes :

```
Temps affichés en msec
n: iter: rec: dict:
5 0.007 0.007 0.014
10 0.008 0.035 0.016
15 0.009 0.355 0.019
20 0.011 3.942 0.021
25 0.012 43.999 0.024
30 0.013 482.903 0.027
100 0.030 ---- 0.061
300 0.078 ---- 0.160
500 0.131 ---- 0.268
700 0.187 ---- 0.384
900 0.245 ---- 0.495
```

Le léger surcoût pour la version "dictionnaire" par rapport à la version itérative est dû au fait que `cpu_time` copie le dictionnaire à chaque appel. Mais on voit clairement que la

croissance est du même ordre ( $O(n)$ ).

Voici les temps avec `cpu_time_without_copy` :

```
Temps affichés en msec
n: iter: rec: dict:
 5 0.002 0.003 0.000
 10 0.003 0.032 0.000
 15 0.004 0.352 0.000
 20 0.005 3.897 0.000
 25 0.007 43.281 0.000
 30 0.008 482.843 0.000
100 0.027 ---- 0.000
300 0.085 ---- 0.000
500 0.146 ---- 0.000
700 0.211 ---- 0.000
900 0.237 ---- 0.000
```

Comprenez-vous pourquoi ?

- La première fois que l'on fait appel à `fibonacci_dict`, toutes les valeurs de la séquence jusqu'à  $n$  sont stockées dans `known`.
- Cela signifie que si on exécute la même fonction à nouveau avec un paramètre  $m \leq n$ , le seul travail à réaliser est d'aller chercher la valeur dans le dictionnaire ( $O(1)$  en moyenne).
- Pour rappel le temps CPU est calculé en réalisant plusieurs séries de tests. Chacune de ces séries réalise un certain nombre d'appels à la fonction à tester. Ensuite, la moyenne des temps de chaque série est calculée. Enfin, seule la meilleure moyenne est utilisée.
- Cela explique que nous avons un temps affiché de 0 msec pour la version dictionnaire (mais c'est de la triche).

Au chapitre 7, nous avons résolu le problème suivant en utilisant une liste.

*Exercice.* Ecrire un programme qui affiche les paires de mots (contenus dans le fichier `words.txt`) qui sont l'inverse l'un de l'autre. Par exemple, `stop` et `pots`.

Son efficacité n'était pas terrible !

*Démonstration en séance :* version améliorée (plus rapide et sans doublons) basée sur un dictionnaire et discussion à propos des complexités des deux méthodes.

## 10.5. Glossaire

*dictionnaire* : séquence mutable de paires "clef-valeur".

*hashable* : propriété d'un type d'objets lui permettant d'être converti de manière déterministe en un nombre entier.

## Tuples

*Un tuple est une séquence immuable • Fonctions avec un nombre variable d'arguments • Tuples comme clefs d'un dictionnaire • Comparaison de tuples • Quelle séquence choisir ? • Glossaire*

---

### 11.1. Un tuple est une séquence immuable

Un *tuple* est une séquence de valeurs. On définit un tuple en spécifiant les valeurs, séparées par des virgules, entre parenthèses<sup>1</sup>.

```
>>> t = ('a', 'b', 'c')
>>> type(t)
<class 'tuple'>
>>> print(t)
('a', 'b', 'c')
>>> t = 'a', 'b'
>>> type(t)
<class 'tuple'>
```

Remarques :

- les valeurs peuvent être de n'importe quel type ;
- les valeurs sont indicées par des entiers ;
- les tuples sont immuables.

```
>>> t = ('a', 1, [1, 2])
>>> t[0]
'a'
>>> t[-1]
[1, 2]
>>> t[0] = 'b'
TypeError: 'tuple' object does not support item assignment
```

Pour distinguer syntaxiquement une unique valeur et un tuple contenant une seule valeur, on doit utiliser une virgule après la seule valeur du tuple.

```
>>> t = ('a',)
>>> print(type(t), t)
<class 'tuple'> ('a',)
```

---

1. L'utilisation des parenthèses est facultative, mais est souvent utilisée pour plus de clarté.

```
>>> t = 'a',
>>> print(type(t), t)
<class 'tuple'> ('a',)
>>> t = ('a')
>>> print(type(t), t)
<class 'str'> a
```

Ce sont donc les virgules (plutôt que les parenthèses) qui définissent syntaxiquement les tuples. Celles-ci sont utilisées par convention (car proche de la notation mathématique des  $n$ -uplets).

Pour créer un tuple vide, on peut utiliser des parenthèses ne contenant rien ou la fonction `tuple` sans arguments.

Si l'argument de la fonction `tuple` est une séquence, elle retourne un tuple avec les éléments de celle-ci<sup>2</sup>.

```
>>> x = ()
>>> y = tuple()
>>> print(x, y, type(x), type(y))
() () <type 'tuple'> <type 'tuple'>
>>> s = 'abc'
>>> t = range(3)
>>> d = {'a':12, 'b':17}
>>> tuple(s)
('a', 'b', 'c')
>>> tuple(t)
(0, 1, 2)
>>> tuple(d)
('a', 'b')
```

### Comparaisons des différents types de séquences (vues)

| type  | mutable? | indices       | type des valeurs | syntaxe         |
|-------|----------|---------------|------------------|-----------------|
| str   | non      | entiers       | caractères       | 'abc'           |
| list  | oui      | entiers       | tous types       | ['a', 'b', 'c'] |
| dict  | oui      | type hachable | tous types       | {'a':4, 'b':2}  |
| tuple | non      | entiers       | tous types       | (a, b, c)       |

Les opérateurs applicables à une séquence immuable avec des indices entiers sont également disponibles sur les tuples, par ex. : accès d'un élément via `[.]`, tranches (slices), concaténation (+), opérateur \*, fonction `len`, boucles `for`, opérateur `in`.

```
>>> t = ('a', 'B', 'C')
>>> t[0] = 'A'
TypeError: 'tuple' object does not support item assignment
>>> t = ('A',) + t[1:]
>>> t
('A', 'B', 'C')
```

### Assignment de tuples

Un tuple de *variables* peut se placer à gauche d'une assignation. Il faut que le membre droit soit un tuple d'expressions ayant le même nombre d'éléments.

```
>>> a = 17
>>> b = 21
>>> (a, b) = (b, a)
>>> print(a, b)
21 17
```

---

2. Dans le cas des dictionnaires, retourne un tuple des clefs.

```
>>> a, b = 1 * 2, 3 + 4
>>> print(a, b)
2 7
>>> a, b = 1, 2, 3
ValueError: too many values to unpack (expected 2)
```

De manière plus générale, le membre droit de l'assignation peut être n'importe quel type de séquence, du moment que le nombre d'éléments correspond.

```
>>> a, b, c = 'xyz'
>>> print(a)
x
>>> uname, domain = 'jack@umons.ac.be'.split('@')
>>> print(uname)
jack
>>> print(domain)
umons.ac.be
>>> key1, key2 = {'a':12, 'b':21}
>>> print(key2)
b
```

Strictement parlant, une fonction ne peut retourner qu'une seule valeur. Utiliser un tuple comme valeur de retour permet de contourner cette limitation.

```
def min_max(t):
 min = max = t[0]
 for k in t:
 if k > max:
 max = k
 if k < min:
 min = k
 return min, max

>>> print(min_max(range(3,8)))
(3, 7)
```

## 11.2. Fonctions avec un nombre variable d'arguments

Une fonction peut prendre un nombre variable d'arguments. Par exemple les fonctions `min` et `max` peuvent être appliquées sur des séquences, mais également sur un nombre indéfini d'arguments.

```
>>> t = [5, 7, 8, 1, 3]
>>> max(t)
8
>>> max(4, 7, 3)
7
>>> min(4, 7, 3, 2)
2
```

Un autre exemple est la fonction `print` dont le nombre d'arguments est libre.

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, quelque soit leur nombre.

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

>>> f(2, 4)
<class 'tuple'> of length 2
(2, 4)
```



```
>>> f('hello', 2.0, [1, 2])
<class '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)
```

### Opérateur `*` : `gather` and `scatter`

*Gather* : Utilisé devant le nom d'un paramètre dans l'en-tête d'une fonction, l'opérateur `*` rassemble les arguments en un tuple.

*Scatter* : 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)
```

## 11.3. Tuples comme clefs d'un dictionnaire

Il est fréquent d'utiliser des tuples comme clefs 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 clefs 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
```

## 11.4. Comparaison de tuples

Les opérateurs de comparaison fonctionnent avec les tuples (et les autres séquences). Python commence par comparer le premier élément de chaque séquence. S'ils sont égaux, il compare l'élément suivant et ainsi de suite jusqu'à trouver le premier élément qui diffère. La comparaison s'effectue sur ce premier élément qui diffère, les autres éléments sont ignorés, quelles que soient leurs valeurs.

On parle d'ordre *lexicographique* (car c'est le même comportement pour des chaînes de caractères).

```
>>> (0, 1, 2) < (0, 3, 1)
True
>>> (0, 1, 20000000) < (0, 3, 1)
True
>>> (0, 1, 2) < (0, 1, 1)
False
```

## 11.5. Quelle séquence choisir ?

Tuples, listes, dictionnaires... quelle séquence choisir pour stocker ses données ?

*Exercice.* Quel type de séquence choisiriez pour :

- stocker les noms des douze mois de l'année ;
- stocker les emails de vos amis ;
- stocker votre playlist du moment.

Un tuple est adapté pour les données qui ne changent pas (ou ne doivent pas changer).

```
months = ('January', 'February', 'March', 'April', 'May', 'June', \
'July', 'August', 'September', 'October', 'November', 'December')
```

Un dictionnaire est adapté pour retrouver rapidement des données à partir d'une clef unique.

```
emails = {'Bill' : 'bill.baroud@gmail.com', 'Joe' : 'john.doe@umons.ac.be'}
```

Une liste est adaptée pour stocker de manière séquentielle des données qui peuvent être modifiées.

```
playlist = ['come_on_cama.mp3', 'twilight_of_the_thundergod.mp3', 'chaos_AD.mp3']
```

## 11.6. Glossaire

*tuple* : séquence immuable d'éléments.

*gather* : opération qui consiste à assembler un argument de taille variable en un tuple (opérateur `*` dans l'en-tête d'une définition de fonction).

*scatter* : opération qui consiste à traiter une séquence comme une liste d'arguments (opérateur `*` lors de l'appel d'une fonction).

---

## Fichiers et exceptions

*Lecture et écriture de fichiers • Sauvegarder des objets • Exceptions • Gérer les exceptions • Créer son propre type d'exception • A lire par soi-même (pour aller plus loin) • Glossaire*

---

### 12.1. Lecture et écriture de fichiers

#### Persistence

Les programmes réalisés jusqu'à présent s'exécutent pour un court laps de temps et produisent des sorties à l'écran. Quand ils se terminent, leurs données disparaissent.

D'autres programmes sont *persistants* : ils tournent longtemps (ou tout le temps) ; gardent une partie de leurs données de manière permanente (sur un disque dur par ex.) ; et s'ils sont interrompus et relancés, ils reviennent là où ils en étaient.

*Exemples* : systèmes d'exploitation, serveurs web, programmes de traitement de texte, etc.

Un moyen simple de conserver des données (objet de ce chapitre) : lire et écrire dans des fichiers "textes" ou sauvegarder l'état du programme dans une base de données simple (module `shelve`).

*Illustration* : exécution du script `rememberYou.py`

#### Lecture dans un fichier texte

Un *fichier texte* est une séquence de caractères stockée sur un media permanent comme un disque dur, un DVD ou une clef USB.

Nous avons vu comment lire un fichier ligne par ligne grâce à la méthode `readline` ou via une boucle `for`.

Script `readLines.py` :

```
import sys
```

```
filename = sys.argv[1]
f = open(filename)

firstLine = f.readline()
print('First line:\n ->', firstLine.strip())

print('Next lines:')

for line in f:
 print(' ->', line.strip())
```

La lecture est *séquentielle*, c-à-d que chaque ligne est lue l’une après l’autre et que chaque appel à `readline` ou chaque itération de la boucle `for` avance la position courante de la lecture d’une ligne dans le fichier.

Fichier `example.txt` :

```
Un
Deux
Trois
```

Résultat de `python3 readLines.py example.txt` :

```
First line:
-> Un
Next lines:
-> Deux
-> Trois
```

Alternativement, on peut lire l’entièreté d’un fichier avec la méthode `read`.

Script `readAll.py` :

```
import sys

filename = sys.argv[1]
f = open(filename)

content = f.read()
print('content:', repr(content))
```

*Remarque* : la fonction `repr` prend n’importe quel objet en paramètre et affiche une représentation de celui-ci sous forme de chaîne. Dans le cas des chaînes de caractères, cela permet de voir les caractères spéciaux (comme `\n`).

Résultat de `python3 readAll.py example.txt` :

```
content: 'Un\nDeux\nTrois'
```

Toutes les méthodes des chaînes (comme `split`) sont alors disponibles pour travailler sur le contenu, mais si le fichier est très grand, il vaut mieux travailler ligne par ligne.

*Remarque* : il existe également la méthode `readlines` (notez le “s”) qui retourne une liste dont les éléments sont les lignes du fichier (chaînes).

Pour réouvrir un fichier (par ex., pour changer le mode “lecture” en “écriture” ou pour revenir au début du fichier), il faut d’abord le fermer avec `close`, puis l’ouvrir à nouveau avec `open`.

```
>>> f = open('example.txt')
>>> content = f.read()
>>> content2 = f.read()
>>> content
'Un\nDeux\nTrois'
>>> content2
''
>>> f.close()
```

```
>>> f = open('example.txt')
>>> content2 = f.read()
>>> content2
'Un\nDeux\nTrois'
```

Un fichier devrait toujours être fermé quand la lecture (ou l'écriture) est terminée, pour pouvoir le réutiliser et / ou le sauvegarder correctement.

```
file = open("words.txt")
cnt = 0
for line in file:
 cnt += 1
file.close()
```

Comme déjà vu dans le chapitre 6, depuis Python 3, utiliser `with open...` permet de fermer le fichier automatiquement à la fin du `with`, même si une exception se produit.

```
cnt = 0
with open("words.txt") as file:
 for line in file:
 cnt += 1
```

### Écriture dans un fichier texte

Pour ouvrir un fichier en mode “écriture”, on doit spécifier le mode :

- soit `w` (write) qui crée un nouveau fichier et l'ouvre en écriture ;
- soit `a` (append) qui ouvre un fichier existant et permet l'écriture à la fin de celui-ci.

*Attention* : si le fichier existe déjà et qu'on l'ouvre en mode `w`, l'ancien fichier est écrasé et son contenu supprimé !

On utilise la méthode `write` pour écrire dans le fichier. Celle-ci n'ajoute pas de caractère de fin de ligne. Au besoin, il faut les spécifier avec `\n`.

Script `write.py` :

```
import sys
filename = sys.argv[1]

with open(filename, 'w') as f:
 f.write('Line 1\n')
 f.write('Line 2\n')

with open(filename, 'r') as f:
 content = f.read()
 print(content)

with open(filename, 'a') as f:
 f.write('New text\n')

with open(filename, 'r') as f:
 content = f.read()
 print(content)
```

Résultat de `python3 write.py test.txt` :

```
Line 1
Line 2

Line 1
Line 2
New text
```

Le fichier `test.txt` est créé et contient les trois lignes de texte.

### Opérateur de format

L'argument de `write` doit être une chaîne de caractères. Le script

```
f = open('test.txt', 'w')
x = 52
f.write(x)
```

va afficher : `TypeError: write() argument must be str, not int`

Solution : convertir les autres types que les chaînes avec la fonction `str` ou en utilisant l'opérateur de format `%`.

```
f.write(str(x))
s = "%.2f" % (1 / 3)
f.write(s)
```

Ici, `s` sera la chaîne `'0.33'`. L'opérateur de format `%` est explicité plus en détails à la fin de ce chapitre (section "à lire par soi-même").

## 12.2. Sauvegarder des objets

### Une petite base de données

Dans ce qu'on a vu précédemment, il faut convertir les données en chaînes de caractères pour les écrire dans un fichier. Ce n'est pas toujours pratique pour sauver, puis récupérer, des objets plus complexes que des chaînes ou des entiers (par ex. listes, dictionnaires, etc.).

Le module `shelve` permet de *sérialiser* des objets, c-à-d, de les sauver, puis de le récupérer en l'état, très facilement.

### Shelve

Comment utiliser `shelve` ?

- la fonction `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 fonction `shelve.close` permet de fermer le `shelve`, comme pour un fichier.

Soit la première session interactive suivante.

```
>>> import shelve
>>> t = list(range(4))
>>> p = (1, 2, 3)
>>> s = 'hello'
>>> db = shelve.open('data')
>>> db['liste'] = t
>>> db['point'] = p
>>> db['chaîne'] = s
>>> db.close()
>>> exit()
```

Un fichier `data.db` a été créé et contient toutes les données du `shelve`. L'extension `.db` est

ajoutée automatiquement.

Lors d'une session suivante :

```
>>> import shelve
>>> db = shelve.open('data')
>>> print(db['chaine'])
hello
>>> print(db['liste'])
[0, 1, 2, 3]
>>> db['float'] = 3.14
>>> for k in db:
 print(k, ': ', db[k])

point : (1, 2, 3)
chaine : hello
float : 3.14
liste : [0, 1, 2, 3]
```

*Illustration : lecture du script rememberYou.py*

## 12.3. Exceptions

Les exceptions sont fréquentes, en particulier quand on manipule des fichiers.

```
>>> 3 / 0
ZeroDivisionError: division by zero
>>> t = range(4)
>>> t[4]
IndexError: list index out of range
>>> s = '2' + 2
TypeError: Can't convert 'int' object to str implicitly
>>> open('xyz.txt')
FileNotFoundError: [Errno 2] No such file or directory: 'xyz.txt'
>>> open('/etc/passwd', 'w')
PermissionDeniedError: [Errno 13] Permission denied: '/etc/passwd'
>>> open('music')
IsADirectoryError: [Errno 21] Is a directory: 'music'
```

### Lancer une exception

Une exception indique que quelque chose d'exceptionnel s'est produit. Vous pouvez vous-même lancer des exceptions.

*Syntaxe :* `raise ExceptionType('message')`

```
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
```

(Illustration en séance)



## 12.4. 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 plutôt effectuer du code spécifique à ce moment ?

*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 du module `os` (voir la section "à lire par soi-même" concernant les noms de fichiers et chemins).

*Solution 1* : utiliser des instructions conditionnelles pour éviter les exceptions.

*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')
IsADirectoryError: [Errno 21] Is a directory: 'music'
```

- comment être sûr de ne pas oublier un cas exceptionnel ?

Il vaudrait donc mieux 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.

*Syntaxe* :

```
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 :*

```
valid_int = False

while not valid_int:
 try:
 x = int(input("Please enter a number: "))
 valid_int = 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.

Les types d'exceptions les plus fréquentes que vous pouvez gérer (ou lancer) :

|                         |                                                                         |
|-------------------------|-------------------------------------------------------------------------|
| <code>ValueError</code> | Erreur de valeur (par ex., ne respecte pas les préconditions)           |
| <code>TypeError</code>  | Erreur de type pour un paramètre                                        |
| <code>IndexError</code> | Indice hors bornes                                                      |
| <code>IOError</code>    | Toutes les erreurs relatives aux fichiers (ou autres entrées / sorties) |

Voir aussi : [docs.python.org/3/tutorial/errors.html](https://docs.python.org/3/tutorial/errors.html)

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())
 print('All is OK with', i)
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-clef `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 ouvert avant le `try`, ou sauvegarder des données, etc.

*Exemple* :

```
try:
 raise KeyboardInterrupt
```

```
finally:
 print('Goodbye, world!')
```

(illustration en séance)

*Remarque* : la commande “Ctrl-C” permet d’interrompre l’exécution d’un programme. En Python, une telle commande provoque un `KeyboardInterrupt`.

*Illustration* : exécution du script `malicious.py`

*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")
```

## 12.5. Créer son propre type d'exception

Pour l’exemple on va créer une exception de type `UmonsError`. Pour créer ce nouveau type d’exception la syntaxe est :

```
class UmonsError(Exception):
 pass
```

On peut maintenant l’utiliser comme n’importe quelle autre exception.

*Illustration* : voir script `umonsError.py`

*Remarque* : on reviendra sur le mot-clef `class` dès le prochain chapitre.

## 12.6. A lire par soi-même (pour aller plus loin)

|                 |                                                                                                                      |
|-----------------|----------------------------------------------------------------------------------------------------------------------|
| <code>%d</code> | Entier (base 10)                                                                                                     |
| <code>%i</code> | Entier (base 10)                                                                                                     |
| <code>%o</code> | Entier (octal, base 8)                                                                                               |
| <code>%x</code> | Entier (hexadecimal, base 16, en minuscules)                                                                         |
| <code>%X</code> | Entier (hexadecimal, base 16, en majuscules)                                                                         |
| <code>%e</code> | Réel, format scientifique (base 10, minuscule)                                                                       |
| <code>%E</code> | Réel, format scientifique (base 10, majuscule)                                                                       |
| <code>%f</code> | Réel (base 10)                                                                                                       |
| <code>%g</code> | Réel (base 10) : format scientifique ou décimal en fonction de l’exposant, n’affiche pas les zéros non significatifs |
| <code>%r</code> | Chaîne (convertit tout objet avec <code>repr</code> )                                                                |
| <code>%s</code> | Chaîne (convertit tout objet avec <code>str</code> )                                                                 |
| <code>%%</code> | Pas d’argument converti, retourne le caractère %                                                                     |

D'autres formats existent : voir <https://docs.python.org/3/library/string.html>

*Exemples :*

```
>>> print('Time is %dh%min%dsec' % (2, 34, 56))
Time is 2h34min56sec
>>> a = 12
>>> print('a is %d (decimal) or %o (octal) or %X (hexadec)' % ((a,) * 3))
a is 12 (decimal) or 14 (octal) or C (hexadec)
>>> print('%e %E %f %g' % ((10**9,) * 4))
1.000000e+09 1.000000E+09 1000000000.000000 1e+09
```

Pour les entiers ou les chaînes, on peut spécifier le nombre (minimum) de caractères utilisés pour réaliser un alignement (avec des espaces ajoutés automatiquement avant ce qu'il faut afficher).

- `%xd` (ou `%xs`) : `x` est un nombre qui signifie : utiliser `x` caractères (minimum) pour afficher le nombre (la chaîne).

Pour les réels, on peut également spécifier la précision affichée.

- `%x.yf` : `x` et `y` sont des nombres signifient : afficher `y` chiffres après la virgule, et utiliser `x` caractères (minimum) pour afficher le nombre.

*Exemple :*

```
>>> import math
>>> def f(n):
 return math.sqrt(math.sqrt(2**n))

>>> print('%3s %12s' % ('n', 'f(n)'))
>>> for n in range(10,111,10):
 print('%3d %12.2f' % (n, f(n)))

n f(n)
10 5.66
20 32.00
30 181.02
40 1024.00
50 5792.62
60 32768.00
70 185363.80
80 1048576.00
90 5931641.60
100 33554432.00
110 189812531.25
```

Les fichiers sont organisés en répertoires (ou dossiers). Chaque programme possède son "répertoire courant" (par défaut, le répertoire dans lequel on lance la commande `python3`).

Le module `os` (`os` = operating system) possède une fonction `getcwd` (`cwd` = current working directory) qui retourne le nom du répertoire courant.

```
>>> import os
>>> print(os.getcwd())
/Users/hmelot/Algo/code
```

Une chaîne comme `/home/username/filename.txt` qui identifie un fichier est appelé un *chemin*.

- un *chemin absolu* commence à la racine (répertoire le plus haut, symbolisé par `"/`) du système. Exemple : `/home/username/filename.txt`<sup>1</sup>

1. Syntaxe Linux ou Mac, sous windows, on remplace `"/` par `"\"`. L'attribut `os.sep` permet d'obtenir le séparateur du système d'exploitation sur lequel tourne le script.

- un *chemin relatif* commence depuis le répertoire courant (il ne commence donc pas par /, il peut être symbolisé par "."). On peut remonter un répertoire avec les symboles "../". Exemples : ../data/file.txt  
./file.txt.

Dans une instruction comme `open('test.txt')` le chemin est donc relatif au répertoire courant. Pour obtenir le chemin absolu d'un fichier, on peut utiliser `os.path.abspath` (path est un sous-module du module `os`).

Quelques autres fonctions des modules `os` et `os.path` :

- `os.path.exists` vérifie qu'un fichier (ou un répertoire) existe ;
- `os.path.isdir` vérifie s'il s'agit d'un répertoire ;
- `os.path.isfile` vérifie s'il s'agit d'un fichier ;
- `os.listdir` retourne une liste de fichiers et de répertoires contenus dans le répertoire passé en paramètre ;
- `os.path.join` prend les noms d'un répertoire et d'un fichier en paramètre et les joint en un chemin complet (en utilisant le séparateur spécifique au système d'exploitation).

### Exemples

```
>>> os.path.abspath('memo.txt')
'/Users/hmelot/Documents/memo.txt'
>>> os.path.exists('memo.txt')
True
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('music')
True
>>> os.listdir(os.getcwd())
['memo.txt', 'music', 'photos']
```

### Exemple

Le script suivant permet d'afficher les fichiers d'un répertoire donné en argument de la ligne de commande, et affiche (récursivement) le contenu des sous-répertoires.

```
import os
import sys

def walk(dir):
 for name in os.listdir(dir):
 path = os.path.join(dir, name)

 if os.path.isfile(path):
 print(path)
 else:
 walk(path)

walk(sys.argv[1])
```

*Remarque* : le module `os` possède une fonction `walk` qui fait presque la même chose que cette fonction.

## 12.7. Glossaire

*fichier texte* : séquence de caractères stockée dans un support permanent (disque dur, CD-ROM, etc.).

*répertoire* : (dossier) un répertoire possède son propre nom et contient une collection de fichiers ou d'autres répertoires.

*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.

*lancer* (une exception) : on lance une exception quand quelque chose d'exceptionnel se produit, via l'instruction `raise`.

*attraper* (une exception) : on attrape une exception dans une clause `except`, pour y exécuter du code spécifique.

## Introduction aux objets

*La notion d'objet • Les attributs d'un objets et les méthodes `__init__` et `__str__` • Définir le comportement d'un objet via ses méthodes • Surcharge d'opérateurs • Remarques et précautions quand on manipule des objets • Un tout petit mot sur l'héritage • Illustrations de la POO • Glossaire*

---

### 13.1. La notion d'objet

#### Paradigmes de programmation

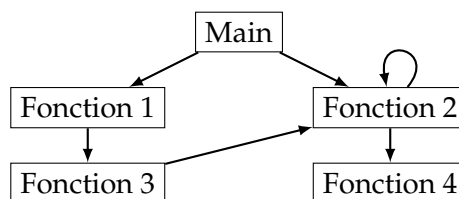
Un *paradigme de programmation* est une *manière* de concevoir et d'approcher la programmation. Python permet d'utiliser différents paradigmes de programmation :

- jusqu'à présent : *programmation procédurière* ;
- dans ce chapitre : *programmation orientée objets*.

#### Programmation procédurière

On détaille pas à pas la liste des instructions permettant de résoudre le problème.

- on découpe le programme en *procédures* (= fonctions en Python) qui s'appellent les unes les autres ;
- on appelle cela également la programmation *impérative*.

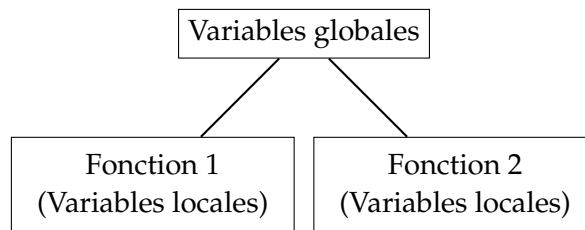


Organisation des *données* en programmation procédurière :

- Les paramètres d'une procédure et les variables définies dans une procédure ne sont accessibles que *localement*.



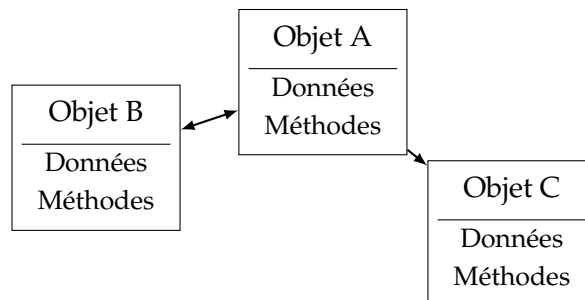
- Il existe aussi des variables accessibles dans toutes les procédures : on parle de variables *globales*.



### Programmation orientée objets (POO)

En POO tout repose sur la notion d'*objet* :

- les fonctions (*méthodes*) sont associées à un certain type d'objets et ne sont accessibles que pour ceux-ci ;
- les données sont "cachées" à l'intérieur des objets (principe d'*encapsulation*) ;
- les objets agissent (et interagissent entre eux) via leurs méthodes.



En réalité, en Python, *tout* est objet : des objets de type entier, chaîne, liste, fonction, exception, dictionnaire, fichier, etc.

Chaque objet possède donc un *type*. En fonction de celui-ci, une série de *méthodes*, d'*attributs* et d'*opérateurs* sont disponibles.

Exemple : objet de type `list`

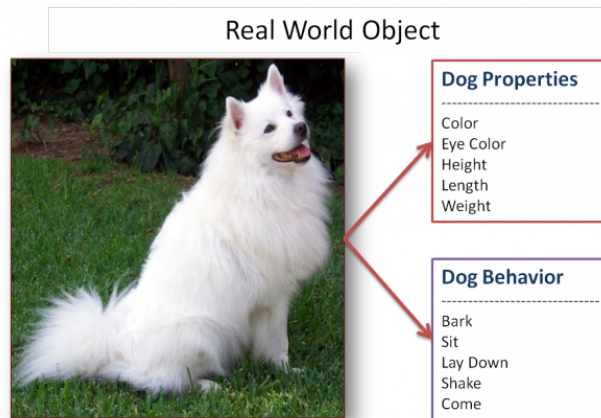
- *attributs* : `__doc__`, ...
- *méthodes* : `pop()`, `sort()`, ...
- *opérateurs* : `[]`, `+`, `*`, ...

Dans ce chapitre, nous allons voir comment définir un nouveau type d'objet via la notion de *classe*.

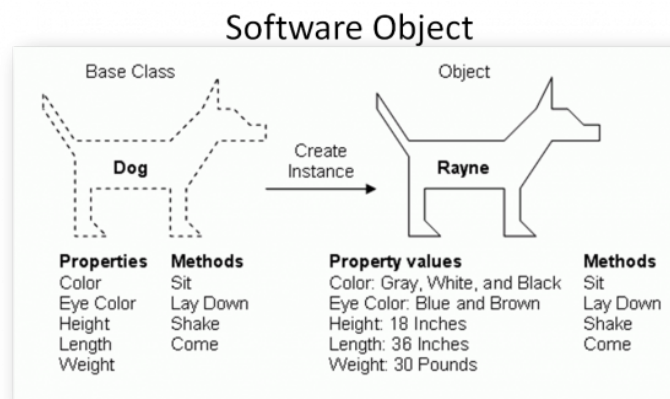
Une *classe* est un modèle décrivant un type d'objets : ses attributs, ses méthodes et ses opérateurs.

On distingue donc les deux notions :

- *objet* : instance (donnée, valeur) d'un certain type ;
- *classe* : modèle décrivant un type et ses propriétés (attributs, méthodes et opérateurs).



Crédits image : <http://lexisbirds.blogspot.be>



Crédits image : <http://lexisbirds.blogspot.be>

*Illustrations :*

- formes géométriques;
- `showRat` et `pyRat` qui utilisent des objets d'une classe `Rational` que vous réaliserez aux TP.

Une série de mécanismes de la POO permet une grande réutilisation du code (exceptions, héritage, interfaces, etc.)

Ces notions sont abordées dans le cours de *Programmation et Algorithmique 2* (B. Quoitin).

A la fin de ce chapitre, nous aborderons brièvement l'héritage.

## 13.2. Les attributs d'un objets et les méthodes `__init__` et `__str__`

Pour modéliser un type d'objets on doit définir leurs *attributs*.

- Les attributs d'un objet modélisent son état, ses particularités dans la famille des objets du même type.
- Concrètement, un attribut est une valeur stockée dans une variable appelée une

*variable d'instance :*

- deux objets d'une classe donnée possèdent des variables d'instances identiques ;
- mais la *valeur* de chacune de ces variables est propre à chaque instance (objet) de la classe.

*Exemple :* tout point dans le plan cartésien possède deux attributs : une coordonnée en *X* et une coordonnée en *Y*. Les points (2,3) et (4,5) sont des objets du même type mais ont des valeurs différentes.

*Question :* quels pourraient être les *attributs* d'un objet de type

- *rectangle* : une largeur, une hauteur, et un coin inférieur gauche (qui est un objet point) ;
- *temps* (durée) : une heure, une minute et une seconde ;
- *étudiant* : un nom, un prénom, un âge, une section, une année d'étude, etc.

Définissons un nouveau type (classe) pour représenter des points dans le plan.

```
class Point:
 """ 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) ;

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__:

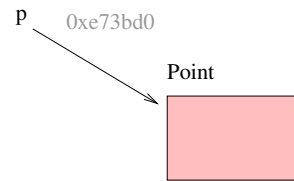
class Point(builtins.object)
 | représente un point dans le plan
 |
 | Data descriptors defined here:
 |
 | __dict__
 | dictionary for instance variables (if defined)
 |
 | __weakref__
 | list of weak references to the object (if defined)
```

Cependant, on peut déjà créer des objets de la classe `Point`. Pour ce faire, on appelle `Point` comme si cela 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).

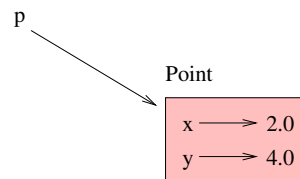
```
>>> p = Point()
>>> print(p)
<__main__.Point object at 0xe73bd0>
```



Pour modéliser et représenter un point dans le plan, on peut utiliser deux attributs : ses coordonnées  $x$  et  $y$ .

Pour assigner un attribut à un objet, on utilise 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`”

```
>>> print(p.x)
2.0
>>> y = p.y
>>> print(y)
4.0
>>> dist_from_orig = math.sqrt(p.x**2 + p.y**2)
>>> print(dist_from_orig)
4.472135955
```

### La méthode `__init__`

Plutôt que de définir directement les attributs d'un objet “à la main” comme présenté ci-avant, on utilise une méthode `__init__` :

- cette méthode `__init__` est une méthode spéciale qui est invoquée automatiquement quand un objet est créé ;
- elle permet de s'assurer que tous les objets d'un même type ont bien les mêmes attributs ;
- elle permet de donner des valeurs spécifiques pour les attributs d'un objet (et de définir des valeurs par défaut) ;
- en POO, on appelle ce type de méthode un *constructeur*.

### Exemple : la classe `Point`

```
class Point:
 """ représente un point dans le plan """

 def __init__(self, x, y):
 self.x = x
 self.y = y
```

### Remarques :

- D'un point de vue syntaxique cela ressemble à la définition d'une fonction mais

on définit `__init__` en plaçant son code à l'intérieur de la définition de la classe.

- Le premier paramètre (appelé `self` par convention) d'une méthode représente l'objet sur lequel la méthode est invoquée. Dans le cas d'un constructeur comme ici, `self` est une référence vers le nouvel objet créé.

#### Exemple : créer un nouvel objet de type `Point`

La méthode `__init__` est invoquée *automatiquement* quand un objet `Point` est créé. Elle permet de spécifier ses valeurs pour `x` et `y` dès la création de celui-ci.

```
>>> p = Point(2, 4)
>>> q = Point(3, 5)
>>> print(p.x)
2
>>> print(q.y)
5
```

Notez que seuls les deux paramètres `x` et `y` sont précisés lors de l'appel au constructeur. En effet, le paramètre `self` est omis : en réalité, `self`, qui est une référence vers l'objet nouvellement créé, est retourné (automatiquement) par le constructeur.

Cette syntaxe peut paraître étrange mais on la comprendra mieux quand on reviendra sur `self` en parlant des autres méthodes.

#### Exemple : la classe `Student`

```
class Student:
 """ represente un etudiant.
 """
 def __init__(self, last, first, age, section, ID):
 self.lastname = last
 self.firstname = first
 self.age = age
 self.section = section
 self.ID = ID

s = Student('Baroud', 'Bill', 21, 'Computer Science', 14023)
```

#### Exemple : la classe `Time`

```
class Time:
 """ represente un temps (heure ou duree).
 attributs: hour, minute, second
 """
 def __init__(self, hour = 0, minute = 0, second = 0):
 self.hour = hour
 self.minute = minute
 self.second = second
```

Remarque :

- Comme pour une fonction, il est possible de spécifier des valeurs par défaut pour les paramètres d'une méthode.

#### Exemple : créer un nouvel objet de type `Time`

```
>>> t1 = Time()
>>> t2 = Time(4, 12)
>>> t3 = Time(18, 35, 17)
```

```
>>> print(t1.hour, t1.minute, t1.second)
0 0 0
>>> print(t2.hour, t2.minute, t2.second)
4 12 0
>>> print(t3.hour, t3.minute, t3.second)
18 35 17
```

*Remarque :*

Dans les exemples précédents, les codes des constructeurs se limitent à initialiser des variables d'instance. En réalité, vous pouvez faire tout autre travail utile pour initialiser un objet.

### La méthode `__str__`

La méthode `__str__` est une méthode qui a pour but de retourner une représentation de l'objet sous forme de chaîne de caractères.

```
class Time:
 # ...
 def __str__(self):
 s = '%02d:%02d:%02d' % (self.hour, self.minute, self.second)
 return s
 # ...
```

Typiquement, la chaîne retournée représente l'état de l'objet, c'est à dire la valeur de (certain) de ses attributs.

Lors d'un appel à `print`, Python affiche automatiquement la chaîne retournée par `__str__` (si celle-ci a été définie).

*Si `__str__` n'est pas défini :*

```
>>> t = Time(8, 30)
>>> print(t)
<__main__.Time object at 0x100704198>
```

*Si `__str__` est défini comme au slide précédent :*

```
>>> t = Time(8, 30)
>>> print(t)
08:30:00
```

Lors d'une conversion d'un objet en chaîne via la fonction `str`, Python fait également appel automatiquement à la méthode `__str__`, si celle-ci a été définie.

```
>>> t = Time(9, 20)
>>> str(t)
'09:20:00'
```

C'est utile pour utiliser les fonctions qui imposent les chaînes de caractères en entrée, comme la fonction `write` par exemple.

Les méthodes `__init__` et `__str__` sont les bases pour un objet :

- `__init__` initialise les variables d'instance ;
- `__str__` permet de les afficher et de convertir l'objet en `str`.

Ces deux méthodes devraient toujours être présentes dans vos définitions de classes.

*Exercice.* Complétez les classes `Point` et `Student`.

Pour le moment

```
p = Point(3, 4)
print(p)
```

donne

```
<__main__.Point object at 0x105450090>
```

car la méthode `__str__` n'est pas définie.

On ajoute dans la classe `Point`

```
def __str__(self):
 s = '(%g, %g)' % (self.x, self.y)
 return s
```

et les instructions

```
p = Point(3, 4)
print(p)
```

produisent maintenant

```
(3, 4)
```

En ajoutant, dans la classe `Student`,

```
def __str__(self):
 s = 'Hello ! My name is ' + self.firstname + ' '
 s += self.lastname + ' and I\'m happy to study ' + self.section
 return s
```

les instructions

```
s = Student('Baroud', 'Bill', 21, 'Computer Science', 14023)
print(s)
```

produisent

```
Hello ! My name is Bill Baroud and I'm happy to study Computer Science
```

## 13.3. Définir le comportement d'un objet via ses méthodes

Les attributs d'un objet modélisent son état, ses particularités dans la famille des objets du même type. Les méthodes spéciales `__init__` et `__str__` permettent d'initialiser et d'afficher ces attributs.

Le *comportement* d'un objet est quant à lui défini par ses autres *méthodes* :

- Pour rappel, une *méthode* est une fonction qui est définie pour un type donné et qui s'applique sur les objets de ce type.
- Un objet possède donc une série de *méthodes* spécifiques à son type (sa classe).

### Exemple : nouvelle méthode pour la classe `Time`

La méthode suivante (ajoutée *dans* la classe `Time`) a pour but de transformer un temps (durée) exprimé en heures / minutes / secondes en une durée exprimée en secondes.

```
def to_sec(self):
 mins = self.hour * 60 + self.minute
 secs = mins * 60 + self.second
 return secs
```

Comme déjà vu avec `__init__` et `__str__` :

- Définir une méthode revient simplement à donner sa définition à l'intérieur de la définition de la classe (attention à l'indentation).
- Dans le code d'une méthode, pour accéder à l'objet sur lequel cette méthode est invoquée, on utilise la première variable nommée `self` (par convention).

Contrairement aux méthodes spéciales (comme `__init__` et `__str__`) qui sont invoquées par Python à certains moments, les méthodes “normales” doivent être explicitement invoquées sur un objet, grâce à la “notation point”.

Nous l’avons déjà fait de nombreuses fois. Exemples : `s.upper()`, `t.pop()`.

Le premier paramètre (`self`) *ne doit pas être mis dans les parenthèses* puisqu’il s’agit de l’objet se trouvant à gauche du point.

```
t = Time(1, 20, 15)
print('Il reste', t.to_sec(), 'secondes de telechargement')
```

produira

```
Il reste 4815 secondes de telechargement
```

Certaines méthodes (appelées *mutateurs*) modifient les attributs d’un objet.

Par exemple, nous allons ajouter deux nouvelles méthodes à la classe `Time` :

- la méthode `update` prend un nombre de secondes en argument et met à jour l’objet de type `Time` sur laquelle elle est invoquée en transformant ce nombre en temps.
- la méthode `add` prend un autre objet de type `Time` en argument et ajoute la durée de cet objet à celle de l’objet sur laquelle elle est invoquée.

```
class Time:
 # ...
 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)
```

Avec ces nouvelles méthodes, nos objets commencent à “prendre vie” et à adopter un comportement qui leur est propre.

```
>>> from mytime import Time
>>> t = Time()
>>> t.update(74700)
>>> duration = Time(1, 35)
>>> print('Le film debute a', t, 'et dure', duration)
Le film debute a 20:45:00 et dure 01:35:00
>>> t.add(duration)
>>> print('Il se terminera a', t)
Il se terminera a 22:20:00
```

### Exemple de mutateur pour la classe `Point`

La méthode suivante (ajoutée *dans* la classe `point`) va déplacer le point en fonction d’un vecteur de direction donné en paramètre.

```
def translate(self, v):
 """ Effectue une translation du point
 en appliquant le vecteur v
 """
 self.x = self.x + v.x
 self.y = self.y + v.y
```

Avec cette nouvelle méthode, nous obtenons :

```
>>> from point import Point
>>> p = Point(3, 4)
>>> print(p)
```



```
(3, 4)
>>> p.translate(Point(2, -1))
>>> print(p)
(5, 3)
```

## 13.4. Surcharge d'opérateurs

### Surcharge d'opérateur

En plus des attributs et des méthodes, on peut 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:
 # ...
 def __add__(self, other):
 res = Time()
 secs = self.to_sec() + other.to_sec()
 res.update(secs)
 return res
 # ...
```

```
>>> from mytime import Time
>>> 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 !

Pour les connaître : <https://docs.python.org/3/library/operator.html>

*Problème.* Pour le moment l'opérateur `+` opère sur deux objets `Time`.

Comment modifier `__add__` pour pouvoir également ajouter un nombre entier (secondes) à ces objets ?

```
class Time:
 # ...
 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
 # ...
```

```
>>> t1 = Time(9)
>>> t2 = t1 + 600
>>> print(t2)
09:10:00
```

*Problème.* Il reste un problème si l'entier est placé avant le `Time`.

```
>>> t1 = Time(9)
>>> t2 = 600 + Time(9)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'Time'
```

Comprenez-vous pourquoi ?

Le problème vient du fait qu'au lieu de demander à un objet `Time` d'ajouter un entier, on fait le contraire : on demande à un objet `int` de lui ajouter un `Time`.

Malheureusement, 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 sera automatiquement invoquée quand un objet `Time` est le membre droit d'un opérateur `+` et que la méthode `__add__` n'existe pas pour un objet `Time` sur le membre gauche.

```
class Time:
 # ...
 def __radd__(self, other):
 return self.__add__(other)
 # ...
```

```
>>> t1 = Time(9)
>>> t2 = 600 + t1
>>> print(t2)
09:10:00
```

Remarques :

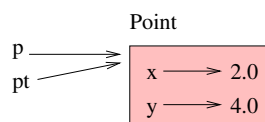
- quand `__radd__` est automatiquement invoquée, le membre droit de l'addition (ici, un `Time`) est placé dans le `self` puisqu'il s'agit de l'objet pour lequel `__radd__` est défini
- les méthodes spéciales peuvent être également invoquées "à la main", comme n'importe quelle autre méthode : c'est ce qu'on fait ici avec `__add__`.

## 13.5. Remarques et précautions quand on manipule des objets

Même si on privilégiera les méthodes en POO, on peut passer un objet en argument d'une fonction. Le paramètre est assigné avec la référence de l'objet, il s'agit donc d'un alias et les attributs de l'objet peuvent être modifiés (les objets sont mutables<sup>1</sup>).

```
def double_point(pt):
 pt.x *= 2
 pt.y *= 2

>>> p = Point(2, 4)
>>> double_point(p)
>>> print(p)
(4, 8)
```

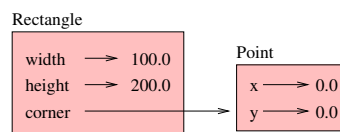


1. Par défaut, les objets créés par le programmeur sont mutables (ce qu'on peut empêcher : ne sera pas vu à ce cours).

Un attribut peut être une référence vers un autre objet.

```
class Rectangle:
 """ represente un rectangle dans le plan
 attributs: width, height, corner
 (coin inferieur gauche, Point)
 """
 def __init__(self, w, h, c):
 self.width = w
 self.height = h
 self.corner = c

p = Point(0, 0)
box = Rectangle(100, 200, p)
```



### Objets comme valeurs de retour

Une méthode (ou une fonction) peut créer un nouvel objet et retourner la référence vers celui-ci. La méthode suivante est ajoutée à la classe `Rectangle` pour en déterminer le point central.

```
def find_center(self):
 center_x = self.corner.x + self.width / 2.0
 center_y = self.corner.y + self.height / 2.0
 return Point(center_x, center_y)
```

### Les instructions

```
box = Rectangle(100, 200, Point(0, 0))
c = box.find_center()
print(c)
```

produiront

```
(50, 100)
```

### Copie d'objets

Un objet passé en argument peut-être modifié par une fonction. Cela peut provoquer des problèmes dans certains cas (cfr. Chap. 7).

On peut copier un objet pour éviter l'aliasing grâce au module `copy`.

```
>>> p1 = Point(3.0, 4.0)
>>> p2 = p1
>>> p2.y = 7.0
>>> print(p1)
(3, 7)
>>> p1 is p2
True
>>> import copy
>>> p2 = copy.copy(p1)
>>> p2.y = 9.0
>>> print(p1)
(3, 7)
>>> print(p2)
```

```
(3, 9)
>>> p1 is p2
False
```

Soit :

```
>>> p1 = Point(3, 4)
>>> p2 = copy.copy(p1)
>>> p1 == p2
```

Que va retourner `p1 == p2` ?

False!

Pour les objets, le comportement par défaut de `==` est le même que l'opérateur `is`. Mais ce comportement peut être modifié en surchargeant l'opérateur `==`.

Dans la classe `Point` :

```
def __eq__(self, other):
 if not isinstance(other, Point):
 return False
 return self.x == other.x and self.y == other.y

>>> p1 = Point(3, 4)
>>> p2 = copy.copy(p1)
>>> p1 == p2
True
```

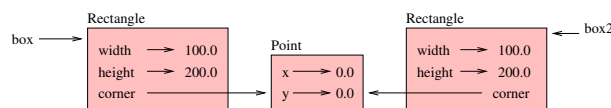
Soit :

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
```

Que va retourner `box2.corner is box.corner` ?

True!

La fonction `copy.copy` réalise une “shallow copy”, c-à-d qu'elle copie les *références* contenues dans les attributs mutables.



Comme vu au chap. 7, la fonction `copy.deepcopy` réalise une “deep copy”, c-à-d qu'elle copie “réellement” 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).

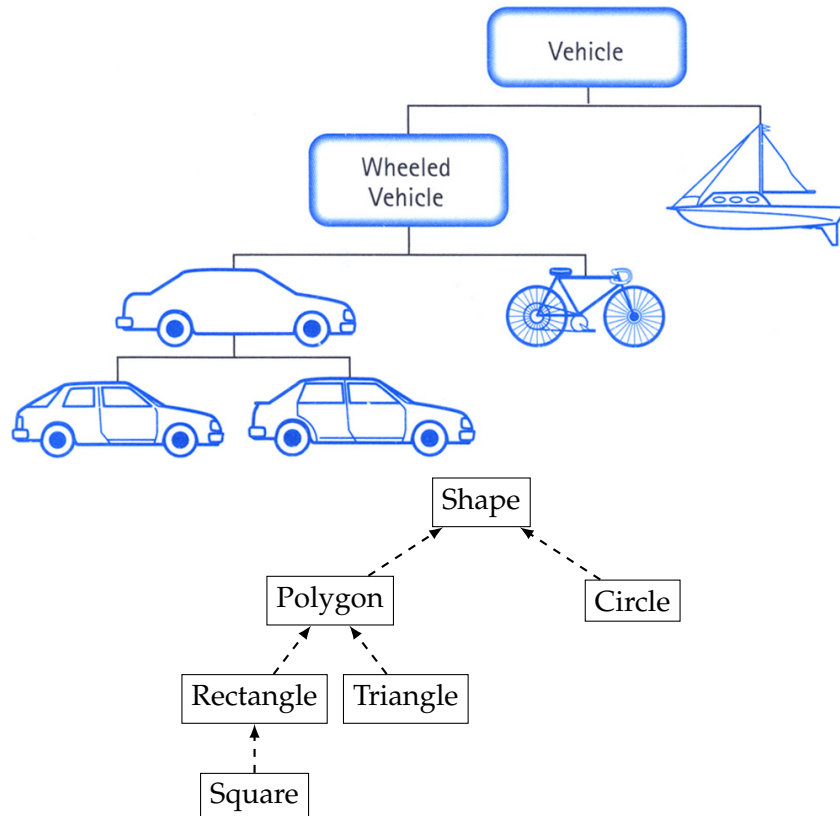
## 13.6. Un tout petit mot sur l'héritage

Une des caractéristiques la plus souvent associée à la POO est l'*héritage*.

L'*héritage* est la capacité de définir une nouvelle classe *N* qui est une version modifiée

(spécialisée) d'une classe existante  $C$  :

- la classe  $N$  hérite automatiquement des méthodes de la classe  $C$  ;
- on peut définir de nouvelles méthodes (spécifiques) pour la classe  $N$  ;
- on peut redéfinir (surcharger) certaines méthode de  $C$  dans  $N$ .



On définit un `Square` en précisant qu'il est un `Rectangle`.

```

class Square(Rectangle):
 """ représente un carré dans le plan
 """
 def __init__(self, l, c):
 """ construit un carré a partir de la longueur l d'un de
 ses côtés et de son coin inferieur gauche c
 """
 self.width = l
 self.height = l
 self.corner = c

```

Toutes les méthodes des `Rectangle` sont accessibles pour un `Square`. Le constructeur a été redéfini (surchargé). Au besoin, on peut définir de nouvelles méthodes uniquement pour les `Square`.

```

>>> from shapes import *
>>> s = Square(50, Point(0, 0))
>>> c = s.find_center()
>>> print(c)

```

(25, 25)

*Présentation rapide en séance du code des formes géométriques. Le code complet est disponible sur Moodle.*

## 13.7. Illustrations de la POO

Présentation rapides

- d’une version POO de l’énigme de la traversée (déjà vue au chapitre 7, avec les listes);
- de l’énigme “othello”.

Les codes complets sont disponibles sur Moodle.

## 13.8. Glossaire

*classe* : une définition de classe permet de définir un nouveau type d’objets.

*instance* : un objet (qui appartient à une classe).

*attribut* : une valeur (nommée) associée à un objet.

*shallow copy* : copie du contenu d’un objet, incluant les références vers d’autres objets imbriqués (mais pas leur copie).

*deep copy* : copie du contenu d’un objet ainsi que des objets qui lui sont imbriqués.

*méthode* : fonction définie à l’intérieur d’une définition de classe, et qui s’applique sur les instances (objets) de cette classe.