# Apprendre à programmer en Python

## pour les jeunes débutants de 7 à 97 ans

Frédéric Laroche

illustrations Gilles Macagno

2015

*Un grand merci à tous les élèves qui m'ont aidé bien involontairement dans l'écriture de ce livre… Sans eux, je n'aurais peut-être même pas eu l'idée de l'écrire.*

# Avant-propos

*L'interface graphique initiée par la célèbre marque à la pomme (en fait le centre de recherche de la société Xerox à Palo Alto, Californie) et devenu un quasi standard obligatoire de nos jours, représente le stade nourrisson de l'interface homme-machine : le nourrisson ne sait pas parler, il montre (pointe) ce qu'il veut et crie (clique) pour l'obtenir. Plus tard il apprend à parler et du coup ses possibilités d'expression s'accroissent infiniment. De même l'utilisateur sérieux d'un système informatique apprend un langage qui lui permet d'exprimer ce qu'il veut faire. La difficulté d'un tel apprentissage est à mon expérience bien plus faible que celle de l'apprentissage de l'anglais par lequel doit passer tout un chacun aujourd'hui.*

*Robert Ehrlich*
*(in http://www.laurentbloch.org/MySpip3/spip.php?article291)*

- Apprendre le « code », apprendre à programmer, apprendre à se servir « intelligemment » d'un ordinateur… fiouuuu ! quel boulot, ça doit être super dur…

Alors là deux réponses possibles viennent de l'assistance :

Première possibilité (le mec qui répond se nourrit de frites et de coca depuis six mois, voix haletante et un peu grasse) :

- Ouais, super hardos, tu veux faire kwa ? Exploser la CB de tes dabs ? Trouver des vidéos piratées sur le net ? Braquer la BCE (Banque Centrale Européenne pour les non initiés) ? Pirater des Bitcoins ? Hacker la Wii de ton petit frère ?...

Deuxième possibilité (le personnage est en costume cravate, sportif, le regard perdu par delà la ligne bleue de l'Oural) :

- Oui, bien sûr, ça me semble intelligent comme démarche, en plus vous pourriez en profiter pour modifier quelques articles protégés de Wikipedia ou lire quelques messages confidentiels sur l'ordinateur de divers personnages peu recommandables, suivez mon regard, dit-il en se tournant vers le premier intervenant...

Là tu craques, tu te lèves et tu dis :

- C'est tout ce qui vous intéresse ? Moi je veux m'AMUSER avec ma machine, pas jouer à l'espion ou au voleur ou Dieu sait quoi ! Apprenez-moi...

Grand silence blanc dans la salle... Un monsieur d'un certain âge, belles moustaches poivre et sel, se lève :

- Écoute jeune fille, l'informatique c'est sérieux, ça demande de faire attention en permanence, d'être tenace et rigoureux, d'apprendre une nouvelle langue avec son vocabulaire, sa grammaire, ses règles et de t'en servir pour communiquer avec le monde entier, machine et humains réunis. Si tu te sens prête, je veux bien t'accompagner un bout de chemin mais il faudra faire quelques efforts intellectuels, ce n'est pas aller sur Facebook ou jouer avec la Xbox...

- Oui, c'est ce que je veux ! Je veux savoir, pas être comme une débile devant ces machines et ne même pas comprendre comment ça marche ! Une voiture, c'est facile : un moteur, des roues, un volant et un frein dans une boite... d'ailleurs la plupart des gens ont leur permis de conduire. Un ordinateur, y'a pas de permis, tout le monde se croit capable de les utiliser sans risque, mais je suis sûre que ce n'est pas le cas !

Le monsieur se lève à nouveau :

- Ok, on va faire comme ça, apprendre en s'amusant. Voilà ce que je te propose : à partir de petits projets on va essayer d'explorer diverses méthodes utilisées pour faire marcher un ordinateur. Je ne vais pas commencer à te faire un cours comme tu as en classe, on va démarrer direct sur des programmes et on verra ce qui t'es nécessaire au fur et à mesure. Ça te va ?

- Yeaaaaah !

- Ah j'oubliais, en fait tu vas avoir deux langues à apprendre en même temps puisque l'informatique s'écrit en anglais la plupart du temps, mais ne t'inquiète pas c'est de l'anglais assez simple et une fois que tu auras compris les rudiments ça te semblera totalement évident… Et puis si vous êtiez deux pour mettre au point vos programmes, ça serait encore mieux.

- Vous avez raison, je vais chercher mon frère jumeau, il est presque aussi intelligent que moi !

Ainsi fut fait !

Quelques choix faits dans l'écriture de ce livre :

Python est un langage très facile d'accès mais néanmoins très puissant ; on peut l'utiliser pendant de nombreuses années, aussi il semble bien adapté pour débuter l'apprentissage de la programmation, disons à partir de 8-9 ans (CE2-CM1), l'âge idéal me semble être 10-11 ans (CM2-6e) afin de connaître les quatre opérations et d'avoir quelques notions de géométrie repérée.

Une possibilité était de partir sur une version simplifiée et « francisée » de Python comme `rur-ple` mais dès que l'on quitte les éléments de base on retombe sur du Python pur jus. Par ailleurs il ne semble pas exister de version pour machines Apple, ce qui est un peu gênant.

Autre choix possible, utiliser uniquement le module `turtle` (*turtle* = tortue en français) comme cela a été fait dans le livre en allemand *Python für Kids*[1] : là encore on reste prisonnier d'un environnement simplifié (et même un peu trop simplifié) et si on veut en sortir, il faut reprendre à la base un nombre conséquent de notions. Ceci dit ce peut être une bonne idée de travailler plus ou moins en parallèle avec la tortue même si c'est un outil un peu dépassé…

Il y a très peu de cours dans le livre au sens propre du terme : la plupart des chapitres à partir du troisième démarrent sur un projet et on se lance à l'eau… Divers éléments sont alors fournis au fur et à mesure pour bien comprendre la démarche

[1] *Python für Kids*, Gregor Lingl, éd. BHV, 2010.

suivie et les outils nécessaires. L'intérêt de cette méthode est évidemment de pouvoir obtenir des résultats très rapidement ainsi que des programmes opérationnels.

Les deux premiers chapitres sont un peu techniques et devront être lus en entier tôt ou tard, même s'ils peuvent être survolés dans un premier temps. Le reste du livre est constitué essentiellement d'une vingtaine de projets à réaliser permettant d'accumuler diverses connaissances techniques ou méthodologiques.

Les chapitres 3 à 6 sont consacrés à la réalisation de projets assez simples ; ils sont numérotés grosso-modo par ordre de difficulté croissante.

Le chapitre 7 explique les principales opérations nécessaires au fonctionnement d'un ordinateur et du calcul en général. Il vaut mieux le lire avant le chapitre 8 sur la cryptographie.

Le chapitre 9 explique le fonctionnement d'Internet et des diverses couches logicielles utilisées.

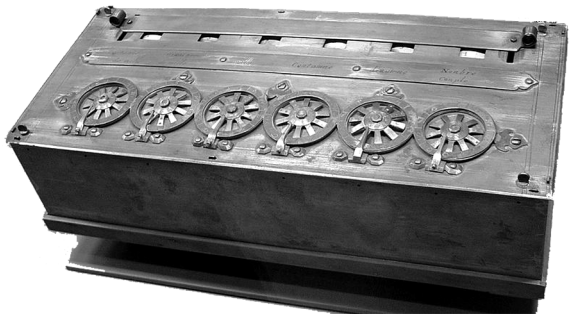
Les chapitres 10 et 11 introduisent à la création graphique à l'aide de la bibliothèque Tkinter : il est indispensable de les lire et surtout de les comprendre avant de passer à la suite. Les chapitres suivants sont des applications diverses de la programmation événementielle et de tout ce qui a été vu précédemment. Ils peuvent être (presque) réalisés dans un ordre quelconque… Le dernier chapitre traite de la manipulation des images et un peu du son : j'espère qu'il est suffisamment clair pour que tu comprennes quelques bases de la révolution numérique…

Un site web dédié te permet de charger les divers programmes sous une forme « à compléter » ainsi que les divers outils nécessaires (voir les détails page 20).

Python est un langage moderne essentiellement **orienté objet** : on trouvera sur Internet une foultitude de programmes utilisant les propriétés de la programmation objet, facilement reconnaissables aux deux mots réservés `class` et `self` qui émaillent lesdits programmes.

J'ai fait le choix de ne pas utiliser ces méthodes et de rester sur des programmes séquentiels classiques tout à fait réalisables et utilisables sans la notion d'objet (classe), bien que dans de nombreux cas on utilise des propriétés d'objets sans le dire. Pour les lecteurs intéressés, le livre *Hello World! Computer Programming for Kids and*

*Other Beginners*[1] utilise le langage des objets très rapidement et peut être une base intéressante pour repartir dans cette direction.



La Pascaline, première machine à calculer mécanique,

construite par Blaise Pascal en 1642.

Musée des Arts et Métiers, Paris. Photographie David Monniaux.

Quelques conventions : en général la multiplication est représentée en informatique par une étoile « * » et la division par la barre oblique « / » (appelé *slash*). C'est ce qu'on va faire dans tout le livre quand il n'y aura pas de confusion possible.

Les mots nouveaux sont en gras, en italique pour l'anglais, ils sont répertoriés dans l'index de fin du livre ; les actions informatiques ou les instructions ou les programmes sont en caractère de machine à écrire[2], généralement sur fond gris : `coucou c'est moi`. Les noms de variables sont en gras dans les commentaires des programmes. Il peut y avoir quelques répétitions entre chapitres, c'est volontaire et destiné à assurer la cohérence de chaque programme et une certaine autonomie aux chapitres.

Cet ouvrage est une première approche de Python et de la programmation en général : j'ai fait ce que j'ai pu pour me limiter aux notions essentielles. De nombreux ouvrages ou cours sur Internet te proposeront bien davantage de possibilités mais

---

1 *Hello World! Computer Programming for Kids and Other Beginners*, Warren & Carter Sand, éd. Manning, 2011.

2 Les plus jeunes d'entre vous seront contents de savoir qu'avant les ordinateurs on utilisait déjà des machines (pour écrire, pour la comptabilité…) d'où sont d'ailleurs issues les diverses dispositions de clavier : pourquoi AZERTY… en français et QWERTY… en anglais ?

comme tu peux le voir on peut faire déjà beaucoup de choses avec des outils de base... et il ne tiendra qu'à toi d'aller plus loin !

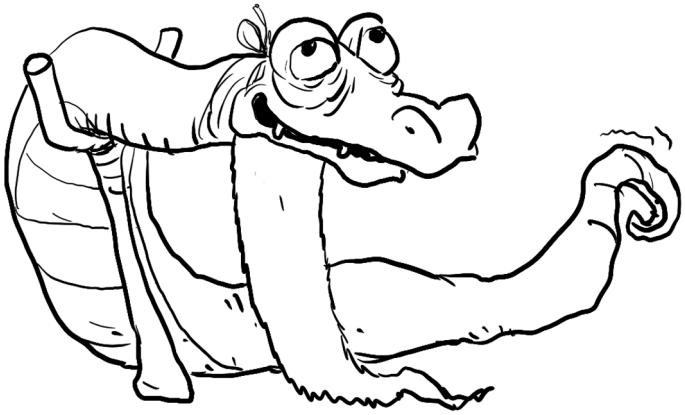
Les illustrations sont dues au talent de Gilles Macagno, les strip de BD viennent de chez Randall Munroe (xkcd.com), la relecture a été effectuée par Éric Zabban, le soutien éditorial a été assuré par Corinne Baud et le soutien moral par mon épouse Christine. Qu'ils soient tous remerciés.

*L'informatique n'est pas la science des ordinateurs,*
*pas plus que l'astronomie n'est celle des téléscopes.*

*Edsger Dijkstra*

On utilise Python dans ce livre, mais tout ce que tu vas apprendre sur la programmation te resservira même avec d'autres langages et même avec des plus compliqués…

Grâce à Python tu auras de bonnes bases pour longtemps et en plus on l'utilise dans les études !



# Table des matières

*Ne vous contentez pas d'acheter un nouveau jeu vidéo. Fabriquez-en un. Ne vous contentez pas de télécharger la dernière application : participez à sa conception. Il ne suffit pas de jouer sur votre téléphone : programmez-le. Personne ne naît « expert en informatique », mais avec un travail régulier et des maths et de la science, à peu près tout le monde peut en devenir un. (...) Les ordinateurs vont être une composante clef de votre futur.*

*Barrack Obama (décembre 2013)*

# Installation de Python et démarrage


## 1.1 Python

### 1.1.1 Un programme c'est quoi ?

L'objectif d'un langage de programmation est de donner des ordres à un ordinateur ou à tout appareil similaire comme les tablettes, les smartphone, les appareils photo numériques, etc.

Le langage de base de l'informatique est constitué d'une série de 0 et de 1 (en réalité du courant électrique), seule possibilité de communiquer directement avec les

machines. Comme il est très compliqué pour un humain d'écrire et de lire des 0 et des 1, on a inventé des langages de programmation qui permettent d'avoir une écriture plus proche de nos capacités. Suivant les langages un **compilateur** ou un **interpréteur** permettent alors de transcrire ce langage en 0 et 1 et de communiquer avec l'ordinateur.

En général on utilise un **éditeur de développement intégré** (EDI) pour écrire les programmes : cet éditeur permet d'écrire les lignes du programme (*lignes de code*) tout en contrôlant la **syntaxe** (la manière d'écrire le programme) ; une fois le programme écrit on l'exécute dans l'EDI et on le corrige (*déboguer* ou *debugger*) suivant les indications de ce dernier.

Typiquement un programme professionnel peut contenir plusieurs milliers de ligne de code voire plusieurs millions… On ne fait pas tout ça d'un seul coup ni tout seul évidemment ! Il y a des équipes avec des chefs de projet, des développeurs, des analystes, etc. qui vont s'attaquer à la fabrication du programme. La réalisation d'un projet peut durer plusieurs années, être remanié, modifié, amélioré… et pourtant il peut rester des bugs ou des failles malgré tout le soin apporté.

Ceci dit, pour apprendre à coder on n'a encore jamais trouvé mieux que d'apprendre en solo et de réaliser soi-même un certain nombre de programmes afin de maîtriser les principes de base, et ceci quel que soit le langage utilisé !

Pour notre part nous ferons des programmes relativement courts (le plus gros doit faire environ 400 lignes de code) mais en essayant d'être sûrs qu'ils fonctionnent bien…

### 1.1.2 Les langages et Python

Il existe une quantité colossale de langages, la plupart créés dans un but bien déterminé : faire du calcul, gérer des données, piloter un drone, naviguer sur Internet, commander une machine à laver, etc. Il existe des langages de bas niveau, très proches du langage directement compréhensible par les machines comme l'**assembleur**, des langages intermédiaires comme le **C** nécessitant beaucoup de travail de la part des programmeurs pour fonctionner dans des environnements divers ou des langages de haut niveau comme **Java** ou **Python** disposant de nombreux outils permettant de travailler rapidement quelle que soit la situation.

Les langages de programmation ressemblent par divers côtés aux langages humains : il y a un **vocabulaire**, souvent constitué de mots anglais (*print*, *while*, *for*, *and*, …), une **grammaire** qui donne les règles à utiliser et une **sémantique** qui permet de donner un

### 1.2.8 Les conditions

Les *conditions* permettent de modifier le comportement du programme en fonction de diverses conditions de ton choix.

Par exemple tu veux changer l'apparence de ton programme en fonction de l'heure : tu peux lui demander d'avoir un fond noir entre 10 heures du soir et 6 heures du matin et un fond blanc le reste du temps.

On pose donc la *condition* : **« S'il fait nuit, je dois dessiner un fond noir, sinon je dessine un fond blanc ».**

On peut considérer qu'il s'agit d'une sorte de dialogue interne au programme lorsqu'on se pose une question :

**Tiens, que se passe-t-il si… et sinon … et enfin…**

ce qui se traduirait en Python par :

**Tiens, que se passe-t-il**

`if` **condition** : *instructions*

`elif` **autre condition :** *instructions*

`else` : *instructions*

(les « : » sont importants, on voit ça un peu plus loin).

Pour écrire les conditions on dispose de l'arsenal suivant :

| action | symboles | action | symboles |
|---|---|---|---|
| si | if | sinon, si / sinon | elif / else |
| égalité | == | différent | ! = |
| inférieur (strict) | < | inférieur (ou égal) | < = |
| supérieur (strict) | > | supérieur (ou égal) | > = |
| inclus | in | pas inclus | not in |
| ou | or | et | and |

Un exemple :

| | |
|---|---|
| `from time import time, localtime` | Appel du module **time** qui contient une fonction **time** et une fonction **localtime.** |
| `t=time()`<br>`h=localtime(t)`<br>`print h[3]`<br>`heure=h[3]` | **time** donne le nombre de secondes écoulé depuis le 01/01/1970 ; **localtime** convertit ce nombre en une liste : [année, mois, jour, heure, minutes, secondes, jour semaine, jour année] |
| `if heure>=22 :`<br>`  print "va te coucher"` | si il est plus de 22 h alors… |
| `elif heure>=15 :`<br>`  print "il faut te lever"` | sinon, s'il est plus de 15 h alors… |
| `elif heure>=13 :`<br>`  print "va à la sieste"` | sinon, s'il est plus de 13 h alors… |
| `else :`<br>`  print "va travailler"` | sinon … on gère tous les autres cas, soit de 0 à 13 h. |

Pour t'entraîner, modifie le programme ci-dessus pour dire qu'entre 12 h et 13 h tu vas déjeuner puis pour lui dire qu'à 7 h tu te lèves et qu'à 8 h 30 mn tu vas travailler.

Autre exemple, je veux peindre l'écran en noir la nuit ou lorsque je fais la sieste ; en français on aurait quelque chose comme ça :

Variable : heure_actuelle (un décimal / *float*)

```
Si (heure_actuelle >= 22.0 et heure_actuelle <= 24.0) ou
(heure_actuelle >= 0.0 et heure_actuelle <= 7.5) ou
(heure_actuelle >= 13.0 et heure_actuelle <= 14.25) alors
Peint_fond(noir) sinon Peint_fond(blanc)
```

Note que Python est assez cool car il permet des choses comme :

```
Si (22.0<=heure_actuelle<=24.0) ou (0.0<=heure_actuelle<=7.5)
ou (13.0<=heure_actuelle<=14.25) alors Peint_fond(noir) sinon
Peint_fond(blanc)
```

### 1.2.9 La logique

Si on reprend le dernier exemple et qu'on veut le traduire sous forme d'instructions Python, on doit utiliser les mots réservés **or**, **and** et **not** qui sont indispensables pour manipuler des conditions variées.

Reprenons cet exemple et écrivons le en Python, on aura quelque chose qui ressemble à ça (après initialisation de la variable `heureActuelle`) :

```
if (22.0<=heureActuelle<=24.0) or (0.0<=heureActuelle<=7.5) or
(13.0<=heureActuelle<=14.25) : PeintFond('noir')
else : PeintFond('blanc')
```

En fait les encadrements comme `(22.0<=heureActuelle<=24.0)` correspondent à un raccourci pour

```
(22.0<=heureActuelle) and (heureActuelle<=24.0)
```

Exemples : quelle est la couleur du fond ?

| heureActuelle | fond | heureActuelle | fond |
|---|---|---|---|
| 8.35 | 'blanc' | 4.43 | 'noir' |
| 13.51 | 'noir' | 7.5 | 'noir' |
| 14.253 | 'blanc' | 23.59 | 'noir' |

Dans la pratique, très souvent les choses sont assez simples et on se contente de tests du genre :

```
jeuFini=False
Tant que not jeuFini : bla bla bla…
```

**jeuFini** est une *variable booléenne* qui ne peut prendre que deux valeurs **True** (Vrai) ou **False** (Faux) : ici il s'agit d'un jeu qui ne s'arrête que si **jeuFini** devient True, donc dans le programme il faudra à un moment que l'on ait l'instruction `jeuFini = True`.

En général on « oublie » dans les tests booléens la valeur **True** : **jeuFini** a la valeur **False** au début, `not jeuFini` a donc la valeur **True** et le programme passe alors à `bla bla bla`.

### 1.2.10 Les boucles

Les boucles ou répétitions permettent d'exécuter une série d'instructions plusieurs fois de suite, ce qui évite en général de dupliquer inutilement des portions de code.

Lorsque l'ordinateur lit le programme et rencontre une boucle, il va exécuter autant de fois que ça lui est demandé le code écrit dans le bloc de la boucle qu'on lui a indiqué.

En gros il y a deux sortes de boucles, les boucles « obligatoires » qui commencent par le mot **for** (*pour*) et les boucles « conditionnelles » commençant par le mot **while** (*tant que*).

| | |
|---|---|
| Boucle FOR | **for** *variable* **in range(***début*, *fin*, *incrément***):**<br>    *instructions* |
| Boucle WHILE | **while** *conditions* **:**<br>    *instructions* |

Une boucle **for** n'est qu'une boucle **while** qui se voile la face…

Les boucles peuvent s'imbriquer les unes dans les autres. Attention, lorsqu'on imbrique des boucles, il faut prendre garde au nom que l'on donne à la variable de chaque boucle. En effet si elles se nomment toutes **k**, le programme va faire de la confiture de boucles, aussi chaque variable de chaque boucle doit avoir un nom propre. Par exemple **i**, **j**, **k**, **n**, ou si elles sont liées à des positions à l'écran ou des dimensions : **x**, **y** et **z**.

### 1.2.11 Les blocs

L'écriture d'une boucle ou d'une condition se termine **TOUJOURS** par deux points ( **:** )

À la suite de ces deux points commence un bloc d'éxécution lequel peut contenir également des boucles ou des conditions.

On peut imbriquer autant de blocs que l'on veut, bien que 2 ou 3 soient en général suffisants. Chaque bloc doit être décalé (**indentation**) sur la droite d'un nombre fixe d'espaces (en général 4 mais ce n'est pas obligé).

On peut également laisser les instructions, séparées par des ; sur la même ligne (`if a > 100 : z = a/10 ; n = a+z ; p=a*n*z`).

Exemple avec **for** :

| | |
|---|---|
| `a = 1` | Affectation (symbole =) de la valeur 1 à la variable **a** et de la valeur 1000 à la variable **maximum**. |
| `maximum=1000` | |
| `L = range(1, 11)` | La variable **L** contient tous les entiers de 1 à 10 : **L** = [1,2,3,4,5,6,7,8,9,10]. |
| `for k in L :` | La variable **k** va prendre toutes les valeurs de **L** les unes après les autres. |
| `    a = a*k` | On multiplie **a** par **k** : **a** vaudra donc 1, 2, 6, 24, 120, etc. |
| `    if a >= maximum :` | Un nouveau bloc avec if / else : après chaque if / else on refait une indentation. |
| `        print "a est trop gros"` | Si **a** est trop gros on arrête le calcul (**break** fait sortir du **for**) et on revient au bloc précédent. |
| `        break` | |
| `    else :` | |
| `        print "a=", a` | |
| `print "Terminé, k=", k` | Retour au programme principal |

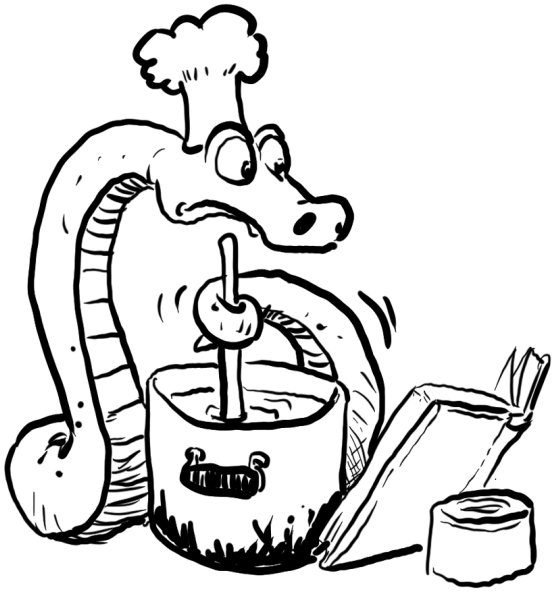
Le même exemple traité avec **while** :

| | |
|---|---|
| `a = 1` | Affectation (symbole =) de la valeur 1 à la variable **a**, de la valeur 1000 à la variable **maximum** et de la valeur 1 à la variable **k**. |
| `maximum=1000` | |
| `k = 1` | |
| `while k<=10 and a<maximum :` | On va augmenter **k** et tester la condition d'arrêt au fur et à mesure. |
| `    a = a*k` | On multiplie **a** par **k** : **a** vaudra donc 1, 2, 6, 24, 120, etc. |
| `    k = k+1` | On augmente **k** de 1. |
| `print "Terminé, k=", k-1` | Retour au programme principal (on enlève 1 à **k**, pourquoi ?) |

Pour t'entraîner : affiche la table de multiplication par 2 (de $2\times1$ à $2\times9$ puis toutes les tables de multiplication de 2 à 6).

Refaire la même chose mais en faisant s'arrêter l'affichage lorsque le résultat est supérieur à 50.

# Algorithmes et recettes de cuisine



Le mot **algorithme** vient du nom du mathématicien arabe Al-Khowarizmi qui vécut vers l'an 800 de notre ère. Il s'agit d'une série de manipulations sur des objets quelconques, aboutissant à un résultat en un temps fini (qui peut aussi bien être très long que très court…).

On peut reprendre la « définition » qu'en donne Donald Knuth, un des plus grands informaticiens de notre temps. Un algorithme doit avoir les propriétés suivantes

**Finitude** : Un algorithme doit toujours se terminer après un nombre fini d'étapes.

**Définition** : Chaque étape d'un algorithme doit être définie précisément, les actions à effectuer doivent être spécifiées rigoureusement et sans ambiguïté pour chaque cas.

**Entrées** : Les quantités qui sont données avant qu'un algorithme ne commence. Ces entrées sont prises dans un ensemble d'objets spécifié.

**Sorties** : Les quantités ayant des relations spécifiées avec les entrées.

**Rendement** : Toutes les opérations que l'algorithme doit accomplir doivent être suffisamment basiques pour pouvoir être en principe réalisées dans une durée finie par un humain utilisant un papier et un crayon.

Un algorithme n'est pas forcément mathématique, même si en informatique c'est le plus souvent le cas…

## 2.1 Tarte aux pommes

### 2.1.1 Un algorithme culinaire

Avec ton papa ou ta maman tu apprends à faire des tartes (pas en mettre, en faire…) ; tu as pris la recette sur un livre de cuisine ou sur Internet et tu t'imagines en robot, essayant de réaliser les différentes actions proposées : tu n'as pas le droit de faire autre chose que ce qui est dans la recette, y arriveras-tu (et surtout est-ce que le résultat sera mangeable) ?

| | |
|---|---|
| Ingrédients | Pâte brisée<br>200 g de farine<br>100 g de beurre à température ambiante<br>1 pincée de sel<br>1 cuillerée à soupe de sucre<br>Garniture<br>5 belles pommes<br>2 cuillerées à soupe de sucre<br>Parfum au choix cannelle ou vanille.<br>Ustensiles<br>1 moule à tarte de 26 cm de diamètre<br>1 casserole<br>1 petit mixer |

| | |
|---|---|
| Préparation | 1. Préparer la pâte brisée en mélangeant la farine, le beurre coupé en petits morceaux, le sel et le sucre.<br>2. Mélanger avec un demi-verre d'eau jusqu'à obtenir une boule et laisser reposer si possible.<br>3. Préparer une compote de pommes : faire cuire dans une petite casserole 2 pommes coupées en petits morceaux avec le sucre, un peu d'eau, la cannelle ou un bâton de vanille fendu pendant 15 mn environ à feu doux ou 8 mn au micro ondes puis mixer la compote.<br>4. Préchauffer le four à 200°.<br>5. Étaler la pâte dans le moule à tarte et la piquer à l'aide d'une fourchette.<br>6. Déposer une couche de compote de pommes sur le fond de la tarte, puis disposer dessus les 3 autres pommes coupées en fines lamelles.<br>7. Saupoudrer de sucre selon son goût.<br>8. Faire cuire à 200°C pendant 30 mn environ. |

Tout d'abord est-ce que tous les ingrédients utilisés ont bien été listés au préalable ?

Est-ce que toutes les actions nécessaires sont bien là ? On peut supposer que certaines actions de base sont déjà bien intégrées par notre robot (mélanger, couper, cuire, etc., ce qui correspond à un système d'exploitation comme Windows ou Linux))

1. Pas de problème.

2. Le demi-verre d'eau n'est pas listé... et puis ça fait combien un demi-verre d'eau ? 10 cl, 50 cl ? Et c'est quoi une boule ? À préciser. Même chose pour les cuillères : une masse serait plus compréhensible (pour un robot).

3. Petite ou grande casserole, ça ne change rien : cette information est inutile. « Feu doux » et « micro-ondes » manquent de clarté : il faut préciser les températures ou les puissances.

4. Durée du préchauffage ? 1 minute ? 1 heure ?

5. Je ne sais pas étaler la pâte dans un moule directement, il me faut d'abord un rouleau, étaler la pâte sur une certaine épaisseur, graisser le moule s'il n'est pas antiadhésif, puis mettre la pâte dans le moule, découper ou rouler les bords... Quand au piquage avec la fourchette notre robot va probablement faire de gros dégâts si on ne lui dit pas comment faire !

6. Ok, quoiqu'il faudrait peut-être dire d'éplucher les pommes ?

7. Le sucre utilisé ici n'est pas prévu dans les ingrédients.

8. Quel est le critère exact d'arrêt ? La couleur ? L'odeur ? Le goût ? 30 mn plus ou moins 5 mn ? Ça peut mal finir !

### 2.1.2 La cuisine algorithmique

La recette de cuisine en informatique s'appelle un **algorithme** : il s'agit d'une succession d'instructions permettant de manipuler des données afin d'obtenir un résultat prédéfini.

Les **données** ici sont les ingrédients, les températures, les temps de cuisson, etc. ; les **instructions** sont les diverses actions que doit faire le robot, actions qui exploiteront elles-mêmes des sous-actions (préchauffer le four c'est vérifier qu'il est vide, l'allumer, mettre la bonne température de réglage, on peut même prévoir une autre action si le four ne démarre pas !).

Comme tu vois, la recette telle qu'elle est donnée ici est incomplète et sur certains points totalement insuffisante… Eh bien en informatique, c'est à peu près la même histoire : on va prendre des ingrédients (données et variables), des méthodes (modules et fonctions) et mixer tout ça pour avoir un script qui marche… Avec une différence de taille par rapport à la cuisine, c'est que tant que ça ne marche pas on peut recommencer certaines parties !

## 2.2 Un langage universel

### 2.2.1 Avec des 0 et des 1

Tu n'as pas eu trop de problème à lire la recette précédente… elle est en français, écrite proprement, tout est dans l'ordre… Trop facile ! Et pourtant, imagine qu'elle soit écrite en espagnol ou en anglais ou même en russe, tu n'y comprendrais pas grand-chose et en russe probablement rien du tout (les Russes écrivent avec un alphabet spécial appelé *alphabet cyrillique* qui vient du grec ancien) !

C'est la même chose en informatique : tout est transcrit dans un langage compréhensible par la machine, constitué de successions de 0 et de 1 :

01111101101000101101100111001000111101011010010000…

pire que du russe ou du chinois !

Alors plusieurs questions peuvent te venir à l'esprit :

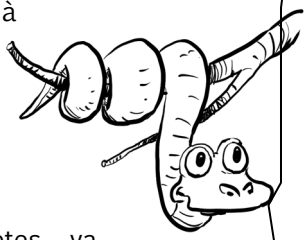
- pourquoi des 0 et des 1 et pas des 2 ou des 5 ou même des symboles tout à fait différents comme ♀ ou Δ ?

- que fait la machine avec ces 0 et ces 1 ?

- est-on obligé d'écrire des 0 et des 1 pour donner des instructions à la machine ?

- les résultats aussi c'est des 0 et des 1 ?

- ça craint...



Pour des explications complètes, va voir en 2.3.1, tu maîtriseras alors nettement mieux le processus.

Gottfried Willem Leibniz était un grand mathématicien du 18e siècle. Il fut le premier à penser que l'on pouvait faire des calculs dans des machines avec uniquement des 0 et des 1 ; il dessina même les plans d'une telle machine mais elle ne fut pas fabriquée.

### 2.2.2 Dans la machine

Un exemple : que se passe-t-il lorsque tu appuies sur une touche (ici une touche numérotée 12) de ton clavier et que la lettre correspondante s'affiche à l'écran dans ton traitement de texte favori « DROW » ?

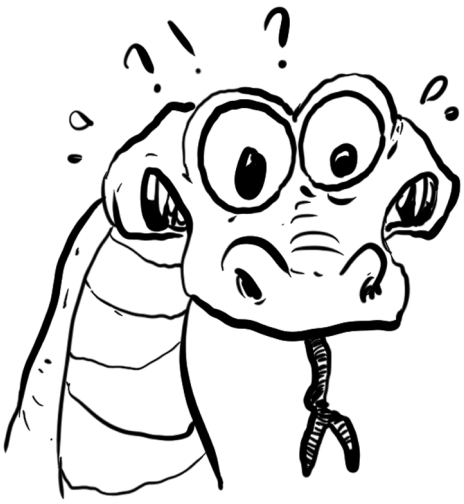
1 - Le clavier transmet au central de connexion (appelé **bus**) de la machine l'information suivant laquelle tu as appuyé sur la touche n°12 : le gestionnaire de clavier (un programme appelé **driver** ou **pilote**) traduit que tu as appuyé sur le caractère n°115 de la table ASCII (voir ci-dessous ; suivant les claviers la touche donnant le caractère n°115 n'est pas forcément au même endroit, c'est le pilote/driver du clavier qui sait où se trouvent les éléments physiques).

2 - Le pilote transmet au bus de communication de l'ordinateur l'information via un **handler** (comme une ligne de téléphone) : le centre nerveux de l'ordinateur, le **processeur**, informe alors la mémoire de stocker dans une zone de mémoire réservée par « DROW » le caractère n°115 et de dire à « DROW » d'afficher le caractère correspondant à l'écran.

3 - « DROW » informe alors, via son handler personnel, la **carte graphique** de dessiner le caractère n°115 à l'emplacement qui convient.

À chaque étape des informations sont transmises, des programmes sont mis en jeu, des réponses sont renvoyées, etc. Toute cette machinerie ne fonctionne qu'avec nos deux symboles qui combinés vont représenter toutes les actions précédentes. Par exemple comment gère-t-on l'affichage de lettres et symboles ?

# Des devinettes faciles et moins faciles...



Bon, initialement c'est un jeu assez simple : l'ordinateur choisit un nombre comme 737 et tu dois deviner ce nombre en lui proposant toi-même des nombres. L'objectif est de manipuler divers outils et d'en faire quelque chose de potable...

## 3.1 La situation de départ

L'ordinateur peut seulement répondre par « le nombre proposé est trop gros » ou « le nombre proposé est trop petit ». C'est un genre de « Juste Prix ».

Dans un deuxième temps on rajoutera un score ainsi que le temps mis pour répondre. Pour terminer on va programmer l'ordinateur pour que ce soit lui qui joue avec une stratégie (optimale) et devine un nombre que tu auras choisi de la manière la plus rapide possible…

En même temps on va programmer graphiquement l'interface de manière à avoir une présentation « professionnelle » (y'a rien qui m'énerve plus que les trucs d'amateur…).

Pour aller plus loin tu pourras faire jouer toute ta famille, ce qui fait que tu devras garder la mémoire des noms et des scores et afficher le meilleur score de la famille !

### 3.1.1 La recette initiale

L'ordinateur choisit un nombre N entre 1 et 1000 par exemple.

Tant que le nombre J proposé par le joueur n'est pas égal au nombre N :

L'ordinateur demande J au joueur :

si J < N : l'ordinateur répond « le nombre J est trop petit » ;

sinon si J > N : l'ordinateur répond « le nombre J est trop gros » ;

sinon : l'ordinateur répond « gagné ».

Fin du tant que.

N est donné d'abord dans le programme pour la mise au point puis on demandera à l'ordinateur de le choisir au hasard par la suite.

Trop facile !

### 3.1.2 Quelques ingrédients

| | |
|---|---|
| Choisir un nombre au hasard entre 1 et 1000. La fonction **randint(a, b)** permet de tirer au hasard un nombre compris entre a et b (a et b compris). | `import random`<br>`u=random.randint(1,1000)`<br>`print u` |
| Saisie d'un entier (mode console) : A est une chaîne qui est convertie en entier avec **int**. Si l'utilisateur ne rentre pas un nombre entier, le programme plante ! | `A = raw_input('Donnez un nombre')`<br>`J = int(A)` |

| | |
|---|---|
| Saisie d'un entier (en utilisant une fonction du module **easyguifr**) : il faut donner davantage de renseignements mais ça vaut le coup, particulièrement si l'utilisateur se trompe (fais l'essai) ! | `import easyguifr as eg`<br>`J = eg.saisie_entier(`<br>`  msg="Donne un nombre"`<br>`  ,titre="saisie entier",defaut=500`<br>`  ,borneinf=1, bornesup=1000)`<br>`print J` |
| Pour les réponses de l'ordinateur tu peux utiliser des instructions `print` et afficher le résultat dans la console ou mieux utiliser la fonction **boite_message** de easyguifr. | `import easyguifr as eg`<br>`if J>N: message='trop gros'`<br>`elif J<N: message='trop petit'`<br>`else : message='gagné'`<br>`eg.boite_message(msg=message)` |

### 3.1.3 Réalisation

On utilise ce qui vient d'être dit :

```
import easyguifr as eg
import random

N=random.randint(1,1000)
J=0
message=''

while J != N:
    J=eg.saisie_entier(msg="Donnez un nombre "
       , titre="saisie entier", defaut=J, borneinf=1, bornesup=1000)
    print J
    if J > N: message=str(J)+' est trop gros'
    elif J < N: message=str(J)+' est trop petit'
    else : message='gagné'
    eg.boite_message(msg=message)
```

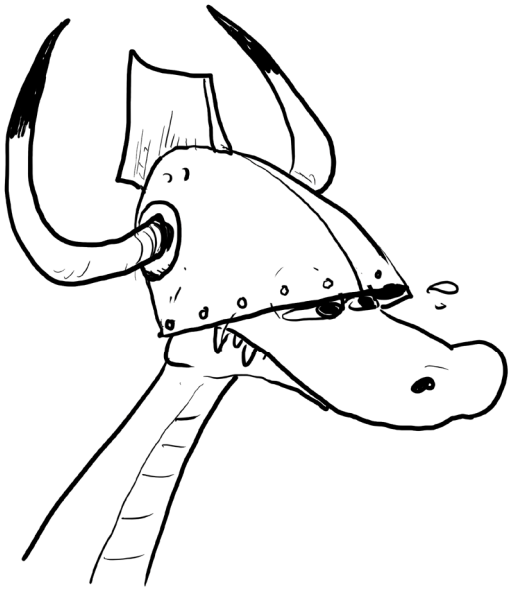
Modules à déclarer.

Initialisation des variables N et J ;

**message** contient une chaîne vide.

```
if J==None : break
else : print J
```

# À la conquête de la division




Ce chapitre n'est pas spécifiquement dédié à la programmation mais il met en lumière quelques mécanismes importants et des fonctions nouvelles comme la fonction *modulo*. On peut très bien s'en passer mais ce serait dommage de ne pas réfléchir sur les outils et techniques présentés.

## 7.1 Un peu de calcul élémentaire

### 7.1.1 Puissances de 10

Il s'agit d'une autre manière d'écrire les nombres de la forme 100000……0000. Comptons le nombre de zéros correspondant au nombre de multiplications nécessaires pour avoir 1 et des zéros :

$1 \times 10 = 10$ c'est 1 multiplication, il y a 1 zéro, on dit que c'est « 10 *puissance* 1 »,

$1 \times 10 \times 10 = 100 = 10^2$ c'est 2 multiplications, 2 zéros, ou « 10 *puissance* 2 »,

$1 \times 10 \times 10 \times 10 = 1000 = 10^3$ c'est 3 multiplications, 3 zéros ou « 10 *puissance* 3 »,

$1 \times 10 \times 10 \times 10 \times 10 = 10000 = 10^4$, 4 zéros ou « 10 *puissance* 4 », etc.

1 million c'est 6 multiplications : 1 000 000 soit $10^6$ ;

1 milliard c'est 9 multiplications : 1 000 000 000 soit $10^9$.

Les nombres 1, 2, 3, 4, etc. sont les *exposants* de $10^1$, $10^2$, $10^3$, $10^4$ : 10 *exposant* 5 c'est comme 10 *puissance* 5 et ça vaut 100 000 (il y a 5 zéros derrière le 1). Remarque que quand on fait 0 multiplication on a 1, soit $10^0$.

Il y a une écriture spécifique des puissances de 10 dans les langages de programmation sous la forme *x* e *n* : 1e5 représente $10^5$ ; 2.36e8 représente $2{,}36*10^8$.

### 7.1.2 Additionner

Pour écrire les nombres on utilise en général les puissances de 10 mais sans le dire la plupart du temps. À gauche de l'égalité suivante on ne les dit pas, à droite on les dit (en fait on ne les dit pas tous, par exemple onze c'est **dix** et un, soixante c'est six **dix**) :

52671 = cinquante deux **mille** six **cent** soixante et onze

que l'on pourrait également énoncer

52671 = 5 **dix mille** 2 **mille** 6 **cent** 7 **dix** et 1

ce qui serait d'ailleurs plus cohérent que la première manière[1]…

On a encore mieux, sous forme détaillée :

[1] Les Chinois, les Coréens et les Japonais écrivent les nombres comme nous mais les énoncent de la deuxième manière, ce qui est beaucoup plus facile à apprendre.

$$52671 = 5 \times 10000 + 2 \times 1000 + 6 \times 100 + 7 \times 10 + 1$$
$$52671 = 5 \times 10^4 + 2 \times 10^3 + 6 \times 10^2 + 7 \times 10^1 + 1$$

Faisons une addition avec deux nombres :
$$\begin{array}{r} 52671 \\ +\ 562 \\ \hline 53233 \end{array}$$

On a été obligé de mettre 1 en retenue lors de l'addition 7 + 6 qui est en réalité 70 + 60 = 130 : la retenue correspond donc au 1 de 130 = 100 + 30, le 100 bascule dans la colonne des centaines.

Pour mieux se rendre compte de la méthode, refaisons l'addition mais en utilisant la forme détaillée et en ajoutant suivant les puissances de 10 :

$$\begin{array}{r} 5 \times 10000 + 2 \times 1000 + 6 \times 100 + 7 \times 10 + 1 \\ +\qquad\qquad\qquad\qquad 5 \times 100 + 6 \times 10 + 2 \\ \hline = 5 \times 10000 + 2 \times 1000 + 11 \times 100 + 13 \times 10 + 3 \end{array}$$

Comme on a 13 = 10 + 3 et 11 = 10 + 1 on obtient également :

$$= 5 \times 10000 + 2 \times 1000 + 10 \times 100 + 1 \times 100 + 10 \times 10 + 3 \times 10 + 3$$

et en regroupant les puissances identiques :

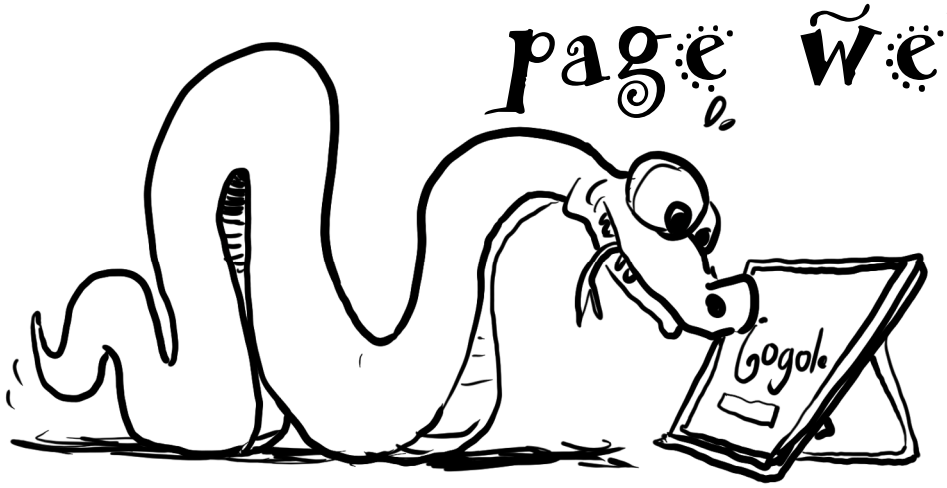
$$= 5 \times 10000 + \overbrace{2 \times 1000 + 10 \times 100}^{3 \times 1000} + \overbrace{1 \times 100 + 10 \times 10}^{2 \times 100} + 3 \times 10 + 3 = 53233$$

Comme tu peux le voir, les retenues dans la manière habituelle de faire les additions en colonne correspondent aux passages de $10 \times 100$ à $1 \times 1000$ et de $10 \times 10$ à $1 \times 100$.

Dans tous les cas, quand on change de puissance de **10** ça veut dire que la somme fait plus de 9 : on dit que l'on calcule en **base 10**. Par exemple quand on est à 9999 et qu'on rajoute 1, on passe à 10000.

Pour calculer en base 10 on a donc forcément besoin de **10** symboles pour écrire tous les chiffres ; regarde comment les formes des nombres se sont transformées au fil du temps :

# Tiens, et si on regardait une page web...




Avant d'aller plus loin avec Python, il me semble intéressant de nous pencher sur une des principales utilisations actuelles de l'informatique (en tout cas au niveau de l'utilisateur standard) : Internet et son fonctionnement.

Ce n'est pas à proprement parler de la programmation Python, mais c'est un thème vaste et important et il ne peut qu'être utile d'en connaître le mécanisme global avec ses diverses facettes. Par ailleurs Python commence à être utilisé dans ce domaine et son rôle dans la programmation web va probablement continuer à se développer assez rapidement.

## 9.1 Schéma général de fonctionnement

### 9.1.1 Adressage

À la base on a deux ordinateurs : un *serveur* et un *client* pour reprendre la terminologie usuelle. Le *client* demande des « choses » au *serveur* qui lui répond en lui renvoyant les choses demandées, sous réserve que ces choses existent et que le *client* ait les autorisations d'accès à ces choses.

En général le *client* est ton ordinateur ou ta tablette ou ton smartphone, bref n'importe quel objet connecté au réseau global (*net* = filet ou réseau en anglais) ; le serveur est en général localisé dans les locaux d'une entreprise appelée *hébergeur* (IBM, Google, Apple, Orange, Bull, Amazon ou moins connu comme OVH ou Gandi).

Le serveur peut également être hébergé chez un particulier (qui peut être toi-même…), la seule condition est d'avoir une **adresse IP** (*Internet Protocol*) **fixe** : à chaque fois que l'on se connecte à Internet on obtient une telle adresse qui peut être invariable (comme 212.27.63.104, adresse fixe de free.fr) ou variable (comme 82.240.120.218 qui est mon adresse dynamique au moment où j'écris ces lignes).

Les serveurs ont toujours une IP fixe de manière à pouvoir être atteints rapidement : une table générale des IP fixes et donc des hébergeurs potentiels est maintenue par l'ICANN avec divers renseignements sur le propriétaire et la localisation. La norme IPv4 n'autorisant qu'un nombre limité d'IP fixes tend à être remplacée par la norme IPv6 qui autorise un nombre colossal d'adresses : tout le monde devrait pouvoir disposer dorénavant d'une IP fixe.

Ci-dessous le classement des pays par nombre d'IP fixes en 2012 (source : CIA)

| | | |
|---|---|---|
| 1 | United States | 505 000 000 |
| 2 | Japan | 64 453 000 |
| 3 | Brazil | 26 577 000 |
| 4 | Italy | 25 662 000 |
| 5 | China | 20 602 000 |
| 6 | Germany | 20 043 000 |
| 7 | France | 17 266 000 |
| 8 | Australia | 17 081 000 |
| 9 | Mexico | 16 233 000 |
| 10 | Russia | 14 865 000 |

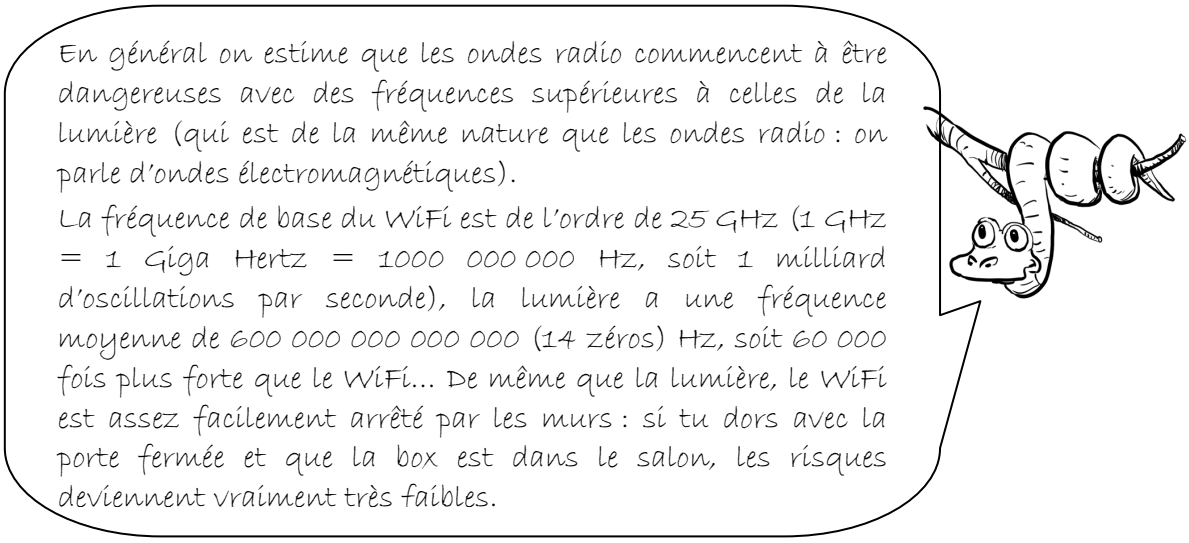
La plupart des adresses IP des serveurs peuvent être converties en un nom de domaine et inversement. Le nom de domaine est plus facilement lisible : **fr.wikipedia.org** est le nom de domaine correspondant à 91.198.174.192. Il s'agit du *système de résolution de noms* (DNS = *Domain Name System* en anglais).

### 9.1.2 Système de communication

Évidemment on ne communique pas comme ça d'un ordinateur à un autre, il y a plein de composants intermédiaires : composants physiques évidemment puisqu'il faut de l'électronique pour pouvoir transmettre les informations et les réceptionner.

Quelques systèmes de transmission habituels : câble **RJ 45** transportant le signal par le réseau **Ethernet** à 100 Mbit/s standard, carte **WiFi** transportant le signal par ondes radio ou même port **USB** ou port **série** (quasiment disparu pour cet usage...).

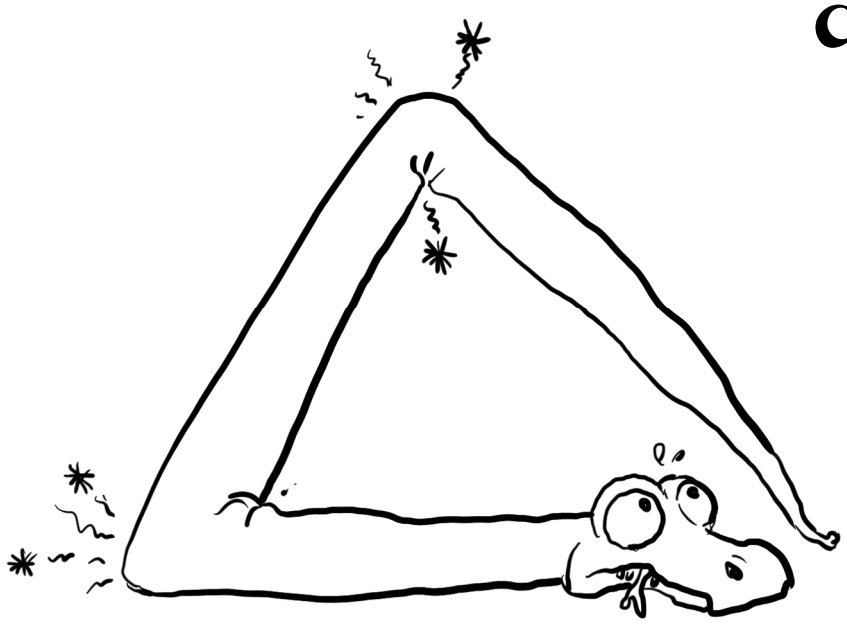
De nombreuses personnes pensent que les réseaux sans-fils WiFi, Bluetooth, GSM (pour les téléphones portables), DECT, etc. sont dangereux pour la santé : pour l'instant aucune étude sérieuse ne semble montrer une véritable influence de ces émissions sur des maladies comme le cancer du cerveau ou les cors aux pieds. Malgré tout ce n'est pas très utile de s'exposer sans raison et garder des écouteurs Bluetooth vissés en permanence sur l'oreille ou le téléphone collé au cerveau !




### 9.1.3 Client - Serveur

Mais surtout il y a de nombreux composants logiciels comme tu vois sur le schéma... Au niveau du client on dispose en général d'informations à transmettre via le navigateur appelées des **requêtes** : quand tu cliques sur un lien ou une image cliquable, quand tu poses une question dans un moteur de recherche, quand tu remplis un formulaire (login, mot de passe), ton navigateur transmet une requête au serveur dont l'adresse correspond à ta demande (adresse en http, IP ou nom de domaine).

# Un peu de dessin



Pour arriver à faire des objets simples ou des animations on a essentiellement besoin du comportement géométrique des divers éléments que l'on va utiliser. Évidemment les manipulations se font essentiellement avec des nombres, comme on va le voir dans ce chapitre.

Les petits scripts qui sont donnés ici servent surtout à te permettre de démarrer et à te montrer quelques applications. Les explications sur les outils de dessin que sont les modules Tkinter et easydessinfr se feront au chapitre suivant.

## 10.1 La mise en œuvre mathématique et informatique

### 10.1.1 Coordonnées

Une idée géniale qu'ont eus les mathématiciens René Descartes et Pierre de Fermat au 17e siècle a été d'attribuer deux nombres à n'importe quel point du plan : un pour chacune des directions du plan, l'horizontale et la verticale, et de s'en servir pour manipuler divers types d'objets du plus simple comme les points ou les segments de droite aux plus compliqués comme les surfaces de l'espace. Les deux nombres attachés à un point s'appelent **coordonnées cartésiennes**.

Ces coordonnées nécessitent deux directions (ou **axes**) munies chacune d'une **unité** de manière à former un **repère** : les coordonnées représentent alors le nombre d'unités nécessaire sur chaque axe pour atteindre un point à partir de l'**origine** ; elles sont notées traditionnellement avec les lettres x et y.

En informatique le plan est en général l'écran ou la fenêtre dans laquelle les affichages vont se faire et on prend habituellement comme repère le bord inférieur pour les x, le côté gauche pour les y, les unités sur chaque axe étant de la taille d'un **pixel**. Sur la figure ci-contre tu as quelques exemples :

- le point A a des coordonnées simples : 160 et 100, ce sont l'**abscisse** et l'**ordonnée** de A. En général on note les coordonnées entre parenthèses avec une virgule (gênant quand les coordonnées sont des nombres décimaux) ou plutôt un point-virgule pour les séparer : A = (160, 100) ou A = (160 ; 100).

- Le rectangle gris en bas commence au point (80 ; 6) passe par (120 ; 6) puis remonte en (120 ; 10) et finit en (80 ; 10).

- Le cercle gris autour de (40 ; 160) est agrandi : les coordonnées indiquées sont celles de deux des coins du carré qui entoure ce cercle : ce seront des coordonnées de ce genre que l'on donnera à Python pour dessiner.

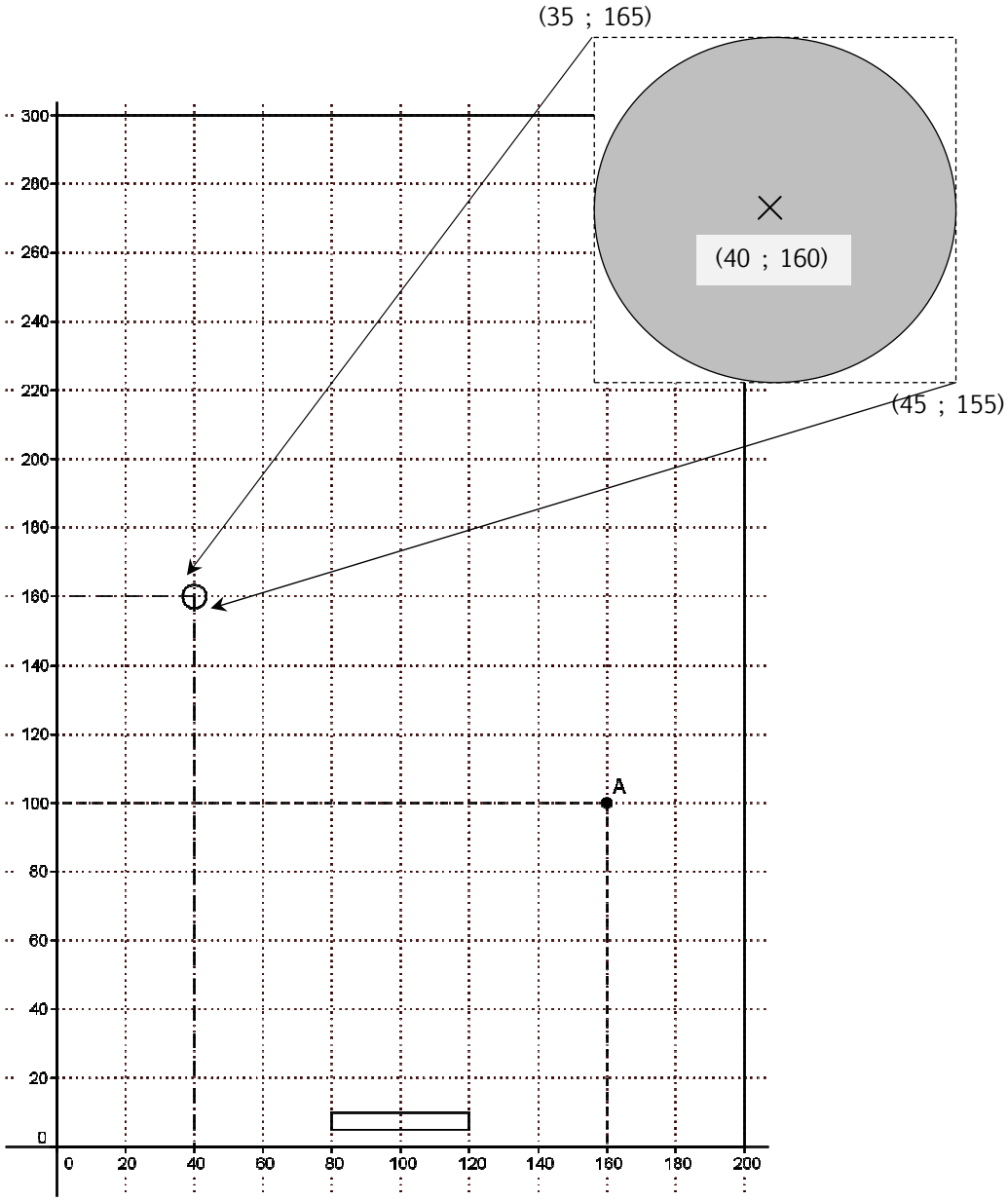
Les coordonnées peuvent être des nombres décimaux : un point peut avoir comme coordonnées (80,7 ; 29,5) ou en écriture informatico-anglo-saxonne : (80.7 ; 29.5).

Entraîne-toi :

- Place les points de coordonnées (180 ; 200) et (60 ; 160).

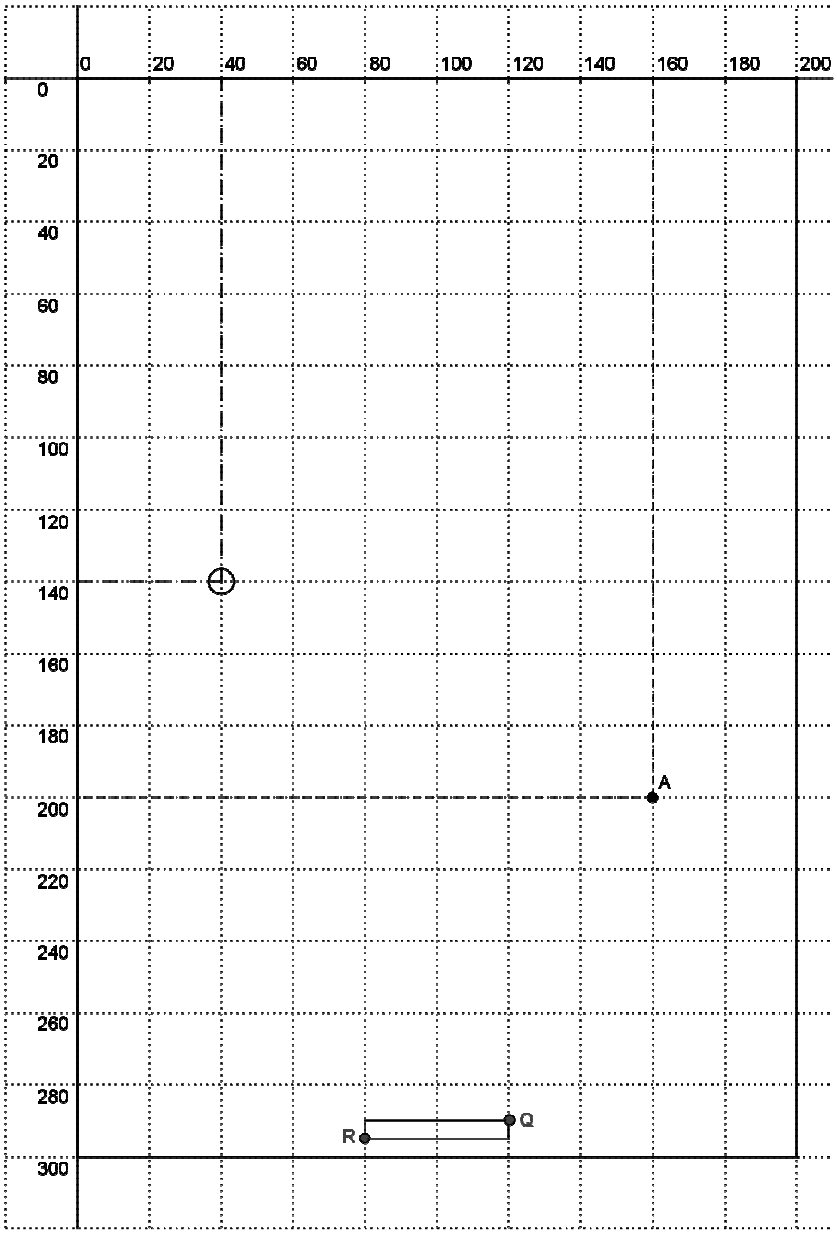
- Dessine le rectangle correspondant à ces points.

- Quelles seraient les coordonnées des sommets principaux du rectangle entourant le cercle de centre A et de rayon 25 ?




Le centre du petit cercle a comme coordonnées (40 ; 160), son rayon est environ 5 : il est inclus dans un rectangle carré de sommets **principaux** (35 ; 165) et (45 ; 155).

Bref, si on veut placer un point quelque part il faut connaître ses coordonnées.




En informatique en réalité, la plupart du temps, les coordonnées d'un écran sont « à l'envers » (par rapport au repère mathématique habituel décrit ci-dessus) pour les ordonnées : on prend 0 en haut et on descend vers le bas ; ça oblige à un peu de gymnastique et à faire attention dans certains calculs.

Autre point important : les coordonnées écran sont des nombres entiers : chaque point physique de l'écran est appelé un **pixel** auquel l'ordinateur donnera une couleur suivant les ordres qui lui sont envoyés.

### 10.1.2 Déplacer un objet

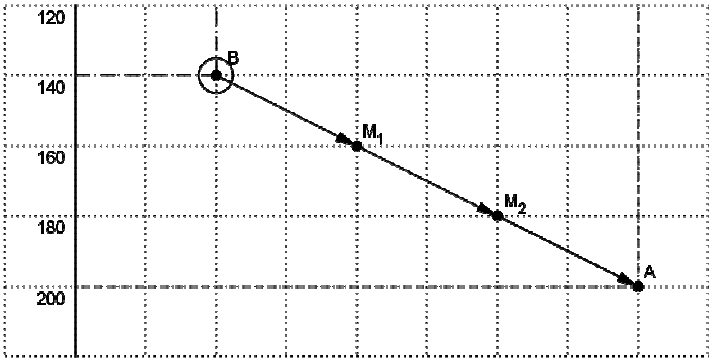
Le principe est semblable à celui du cinéma : on a un objet (par exemple un disque) et on le fait apparaître à des positions successives suivant un « **timecode** ».

Prenons le cercle dont le centre est en B, de coordonnées écran (40 ; 140) : si on veut le déplacer en A c'est facile, il suffit de lui donner les coordonnées de A mais si on veut le voir se déplacer depuis B jusqu'à A c'est un peu plus compliqué ; en plus la manière de se déplacer peut être très variée…

Regardons tout d'abord le déplacement sur un segment de droite.

Appelons M le point qui se déplace, initialement en B, et notons **x** et **y** ses coordonnées ; ajoutons une variable **t** qui représentera le temps : quand **t** = 0 (début du mouvement) on sera en B, quand **t** = 3 on sera en A (tu te demandes pourquoi 3, tu vas voir…)

| **t** | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| **x** | 40 | 80 | 120 | 160 |
| **y** | 140 | 160 | 180 | 200 |




Dans cet exemple on effectue 3 sauts pour aller de B à A : si entre chaque saut on met 1 seconde, on mettra au total 3 secondes.

Chaque saut consiste à faire 40 en avant (**x** devient **x** + 40) et 20 en descendant (**y** devient **y** + 20) : on dit qu'on fait trois fois le **vecteur** (40 ; 20). Ces trois sauts ajoutés correspondent en fait à un unique **vecteur** de B vers A qui a pour coordonnées (120 ; 60) : 120 = 160 – 40 et 60 = 200 – 140 ; on dit que le **vecteur** $\overrightarrow{BA}$ (BA avec une flèche) a pour coordonnées (120 ; 60).

Suppose maintenant que tu veuilles afficher davantage de positions, par exemple 30 positions entre B et A : il te faudra effectuer 30 sauts et à chaque fois te déplacer du vecteur (4 ; 2) car 30*4 = 120 et 30*2 = 60.

Si tu mets toujours 1 seconde pour faire un saut, tu mettras alors 30 secondes pour aller de B vers A, alors qu'avec 3 sauts en mettant 3 secondes tu allais nettement plus vite.

En fait c'est ennuyeux, le temps mis au total ne devrait pas être pris en compte, ou alors il faudrait prendre 0,1 seconde par saut avec 30 sauts, ce qui donnerait bien nos 3 secondes. Évidemment c'est un peu compliqué, aussi on fait les choses plus simplement.

Voici un premier algorithme pour aller d'un point A vers un point B (on n'est plus dans l'exemple) :

- on connaît les coordonnées de A : xA et yA et de B : xB et yB ; M a pour coordonnées x et y ;

- on démarre avec M en A (t = 0) et on finit avec M en B (t = 1) ; v est la « vitesse » souhaitée (en fait ce n'est pas une vitesse mais ce n'est pas grave).

```
x, y = xA, yA ; t = 0 ; v = 100.0
dx, dy = (xB - xA)/v, (yB - yA)/v
dt = 1/v

while t < 1 :
    x, y = x + dx, y + dy
    t = t + dt
```

v = nombre de sauts entre A et B

dt = durée entre deux sauts.

Le vecteur support a pour coordonnées dx et dy que l'on rajoute aux coordonnées de M à chaque saut.

Voici un script qui marche et déplace un disque sur un segment :

```
import easydessinfr as ed
import Tkinter as Tk

def bouge():
  global x, y, t
  if t < 1:
    x, y = x+dx, y+dy
    t = t+dt
```

Modules nécessaires au dessin.

Fonction permettant le déplacement tant que le temps est inférieur à 1.

Modification des coordonnées, avancement du temps,

| | |
|---|---|
| `    can.coords('FL',x-r,y-r,x+r,y+r)` | modification des coordonnées du cercle avec **can.coords**() |
| `    fen.after(int(dt*1000), bouge)` | attente en millisecondes puis reprise de **bouge**. |
| `fen=Tk.Tk()` | Fenêtre pour dessiner, puis définition d'un cadre et d'un bouton de sortie (voir plus loin 11.1.1) ; |
| `can, bouton=ed.cadre_dessin_base(`<br>`    fenetre=fen,large=500,haut=500`<br>`   ,fond='ivoire',nombre_col=10`<br>`   ,nombre_ligne=10, Okbouton='Quitter')` | |
| `xA, yA = 100, 0` | coordonnées des extrémités du segment, |
| `xB, yB = 500, 500` | |
| `t=0; v=200; dt=1/v;` | paramètres du mouvement, |
| `r=20` | rayon du cercle, |
| `x, y = xA, yA` | point de départ de M, |
| `dx ,dy = (xB-xA)*dt, (yB-yA)*dt` | vecteur de déplacement dx, dy, |
| `ed.cree_cercle(cadre = can`<br>`         ,x_C=xB, y_C=yB, rayon=r`<br>`         ,fond='vert',trait='orange'`<br>`         ,nom='FL')` | création du cercle dans le cadre **can** : note le **nom** qui est une chaîne de caractères ; |
| `bouge()` | lancement du mouvement, |
| `fen.mainloop()` | attente des événements. |

Pour voir comment ça fonctionne tu peux modifier les coordonnées de A et B, changer la valeur de v : que se passe-t-il si v augmente ou diminue ? Explique…

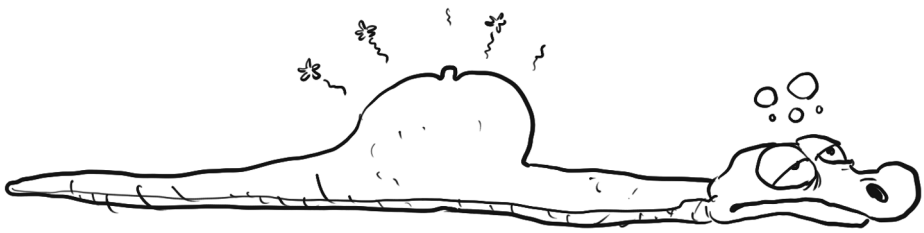
Une remarque importante : l'instruction

```
fen.after(int(dt*1000), bouge)
```

est tout à fait spéciale puisqu'elle se situe à l'intérieur de la fonction `bouge` et qu'elle réutilise `bouge` : on dit que c'est une fonction **récursive**, cette méthode nous resservira très souvent dès qu'on aura besoin d'un mouvement.

Un deuxième algorithme consiste à calculer exactement les coordonnées **x**, **y** de M en fournissant la valeur de **t**. Ça ne change pas grand chose ici mais pour un déplacement sur autre chose que des segments c'est important.

# Un Serpent(in)



Nous allons réaliser un jeu bien connu, appelé en d'autres contrées « Snake » et dans nos beaux pays francophones « Serpent »…

Le principe en est assez simple puisqu'il consiste à faire se déplacer quelque chose qui ressemble à une tête de serpent (symbolique), laquelle va se précipiter sur des pommes qui apparaissent en divers endroits de l'écran. Chaque fois que le serpent mange une pomme, il s'allonge jusqu'à atteindre une taille prédéterminée, moment où le jeu s'arrête (ou change de niveau).

Le jeu ainsi présenté n'est pas d'un intérêt mirobolant mais il présente quelques difficultés techniques que l'on peut réutiliser dans d'autres situations ; par ailleurs on peut facilement le prolonger et créer des niveaux supplémentaires assez facilement.

## 21.1 Eléments de base

### 21.1.1 La recette initiale

Positionnement de la tête du serpent et de la pomme ;

score = 0, nombre d'éléments du serpent = 1, nombre_max d'éléments = 10.

Tant que la partie n'est pas finie :

la tête du serpent se déplace en réaction aux actions de l'utilisateur ;

si le serpent sort du cadre :

la partie est finie (perdue) ;

si la tête du serpent touche la pomme :

le score est augmenté,

la longueur du serpent augmente de 1,

si le score est supérieur au score maximum ou si la longueur du serpent est supérieure à nombre_max, la partie est finie (gagnée) ;

la pomme disparaît et réapparaît ailleurs.

### 21.1.2 Quelques ingrédients

Par rapport à ce qu'on connaît trois points sont importants :

- Le déplacement de la tête du serpent en fonction des actions du joueur : on va utiliser la fonction `canevas.bind` avec certaines touches du clavier : les flèches haut (`'<Up>'`), bas (`'<Down>'`), droite (`'<Right>'`) et gauche (`'<Left>'`).

Chacune de ces actions entraînera la modification d'un vecteur **direction** représenté par une liste de deux nombres : **[x, y]** qui dira ce qu'il faut faire au moment du déplacement.

- La rencontre entre la tête du serpent et la pomme : il existe une fonction de Tkinter qui permet de savoir si des points d'un objet sont communs aux points d'un autre objet.

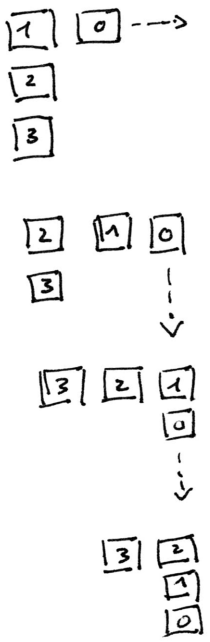
```
z = canevas.find_overlapping(x0,y0,x1,y1)
```

renvoie dans **z** une liste des objets ayant au moins un point dans le rectangle de sommets de coordonnées (x0, y0) et (x1, y1). Il suffit alors de voir si la tête du

serpent est dans la liste ou pas (attention il faut utiliser les Identifiants et pas les Tags mais là encore il y a une fonction qui fait ça très bien). On peut aussi faire le test soi-même mais la première solution est bien plus efficace et donne davantage de souplesse.

- Troisième point, le plus délicat : comment augmenter la taille du serpent, que tout aille dans la bonne direction et comment faire se déplacer TOUT le corps du serpent…

L'astuce consiste à faire comme dans un train : entre deux positions successives du train[1] le wagon numéro n va prendre la place du wagon numéro n – 1. Un petit dessin pour voir :



Les wagons viennent prendre la place de leur prédécesseur : 1 en 0, 2 en 1, 3 en 2 ; on déplace alors 0 dans la direction indiquée.

Si on veut rajouter un élément (4) on le met avant les autres déplacements à la place de l'ancien dernier élément (3), puis on déplace tout le monde sauf 4.

En Python on pourrait se dire qu'il suffit de changer les noms des éléments : par exemple le premier élément s'appelle `'s0'`, on peut l'appeler `'s1'`, `'s1'` s'appelerait `'s2'`, etc.

En fait ça ne marche pas du tout comme ça, il y a un repérage autre des divers objets, interne à Python, et ça ne concordera pas ; il faut changer les coordonnées de chaque élément : les coordonnées de `'s1'` vont dans `'s0'`, celles de `'s2'` vont dans `'s1'`, etc. et `'s0'` prend des coordonnées toutes neuves dépendant uniquement de sa direction.

## 21.2 Le script

### 21.2.1 Le listing

Les choses simples :

[1] Le timing est donné par un intervalle de temps durant lequel un wagon prend la place de son prédécesseur…

```
import easydessinfr as ed
import Tkinter as Tk
from random import randrange
import winsound as ws

def perdu_gagne(gagne):
  global flag
  flag = 0
  can.delete(Tk.ALL)
  ed.cree_image(cadre=can,nomimage="kaa.gif"
              ,x0=x,y0=y,trait='',nom='kaa')
  ed.cree_texte(cadre=can,x0=x,y0=y,texte=""
      ,police=('Georgia',18),trait='rouge'
      ,nom='Tscore')
  ed.cree_texte(cadre=can,x0=x-50,y0=y+50
           ,texte="",police=('Georgia',18)
           ,trait='rouge',nom='Tfin')
  if gagne:
    T = 'Bravo, >Gagné..\nTu peux recommencer'
  else: T = "Dommage... \n\n  Recommence !"
  can.itemconfig('Tscore'
            ,text="Score : "+str(score)+'\n')
  can.itemconfig('Tfin',text=T)
```

Les modules. On peut faire un peu de bruit avec winsound ou pygame (voir chap. 12)

**gagne** est un booléen.

**flag** dit si on peut bouger (flag=1) ou pas (flag=0).

Apparition d'un message lorsque l'on perd ou gagne la partie et nettoyage des objets du cadre **can**.

Regarde les possibilités pour changer de police de caractères sur http://tkinter.fdex.eu/

article *les polices de caractère*.

Note : j'ai utilisé le mot anglais *flag* contrairement à ma résolution de ne mettre que des noms français car c'est un mot très souvent utilisé en informatique. Il signifie simplement *drapeau*, comme on en utilise dans les courses automobiles ; ce mot est souvent utilisé pour des variables booléennes.

Le mouvement :

```
def bouge():
  global long_serpent,score
```

Fonction principale gérant les déplacements

```
if flag != 0:
  u = can.coords('pomme')
  #p0=(u[0]+u[2])/2;p1=(u[1]+u[3])/2
  za = can.find_withtag("s0")
  v = can.coords('s0')
  #s0=(v[0]+v[2])/2;s1=(v[1]+v[3])/2
  zz=can.find_overlapping(u[0],u[1],u[2],u[3])
    if za[0] in zz:
      #if s0-taille_p<=p0<=s0+taille_p and
      #s1-taille_p<=p1<=s1+taille_p:
      ws.PlaySound("velo.wav", ws.SND_ASYNC)
      w = can.coords('s'+str(long_serpent-1))
      ed.cree_rectangle(cadre=can
   ,x0=w[0]+direct[0]*dx,y0=w[1]+direct[1]*dy
   ,x1=w[2]+direct[0]*dx,y1=w[3]+direct[1]*dy
   ,fond='bleu',trait='rouge'
   ,nom='s'+str(long_serpent))
      long_serpent = long_serpent+1
      score = score+100
      xx = randrange(5,2*x-5)
      yy = randrange(5,2*y-5)
      can.coords('pomme',xx-taille_p
        ,yy-taille_p,xx+taille_p,yy+taille_p)
  for k in range(1,long_serpent):
    w = can.coords('s'+str(k))
    can.coords('s'+str(k),v[0],v[1],v[2],v[3])
    v = w
  can.move('s0', direct[0]*dx, direct[1]*dy)
  v = can.coords('s0')
  s0 = (v[0]+v[2])/2 ; s1 = (v[1]+v[3])/2
```

et les collisions.

p0, p1 sont les coordonnées du centre de la tête du serpent mais on peut utiliser la méthode de Tkinter : `find_overlapping`.

Si on veut tester soi-même…

Un peu de bruit…

**w** = coordonnées du dernier élément du corps du serpent qui sert à créer un nouveau segment en tenant compte de la direction actuelle. Attention au nom de ce segment.

Modification des variables importantes.

Nouveau placement de la pomme.

Gestion du mouvement par échange des coordonnées des éléments du serpent.

Et enfin déplacement du serpent dans la direction **direct**.

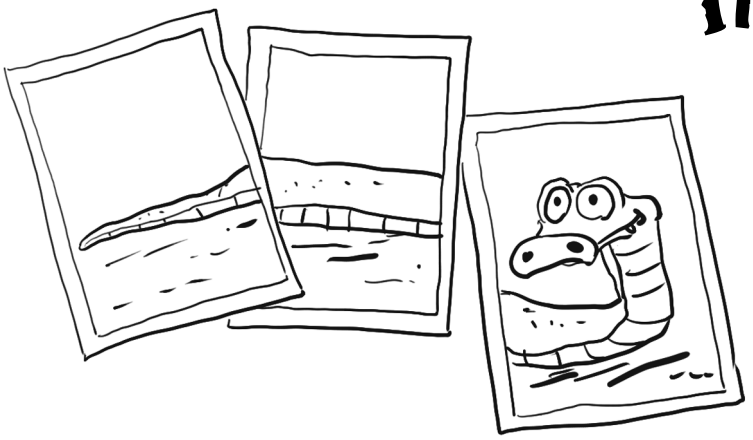
| | |
|---|---|
| `if not(5 < s0 < 2*x-5 and 5 < s1 < 2*y-5):`<br>`  perdu_gagne(False)`<br>`elif score >= score_max:`<br>`  perdu_gagne(True)`<br>`else:`<br>`  tex1.config(text="Score : "+str(score))`<br>`  fen.after(duree, bouge)` | On regarde si le serpent est bien dans le cadre (si non : perdu)<br>ou si on a gagné<br>ou si rien de spécial, auquel cas on affiche le score. Mouvement après 60 millisec. (**duree**). |

Pour lancer une nouvelle partie : l'intérêt est que ça peut servir de porte d'entrée pour un niveau différent.

| | |
|---|---|
| `def nouveauJeu():`<br>`  global flag,score,long_serpent,direct`<br>`  tex1.config(text="Démarrer pour commencer")`<br>`  can.delete(Tk.ALL)`<br>`  long_serpent = 1`<br>`  score = 0`<br>`  direct = [1,0]`<br>`  ws.PlaySound("enclume.wav",ws.SND_ASYNC)` | On peut avoir un nouveau jeu au début ou quand on a perdu ou quand on a gagné et cliqué sur …. |
| `  ed.cree_rectangle(cadre=can`<br>`  ,x0=x,y0=y,x1=x+taille_s,y1=y+taille_s`<br>`  ,fond='orange',trait='rouge',nom='s0')` | La tête du serpent au milieu du cadre. |
| `  ed.cree_cercle(cadre=can`<br>`  ,x_C=randrange(5,2*x-5)`<br>`  ,y_C=randrange(5,2*y-5)`<br>`  ,rayon=taille_p,fond='vert'`<br>`  ,trait='bleu',nom='pomme')` | La pomme quelque part au hasard dans le cadre. |
| `  if flag == 0: flag = 1`<br>`  bouge()` | Le jeu (re)démarre et (re)lance le mouvement. |

L'instruction `can.delete(Tk.ALL)` efface tous les objets construits dans le cadre **can**. C'est un peu violent comme méthode mais on est sûr de ne pas oublier un truc dans un coin.

# Multimédia, comment ça marche ?



Ce dernier chapitre va nous permettre de travailler de manière plus approfondie sur le thème de l'image : chargement, sauvegarde et surtout manipulations…

Les images numériques sont constituées de lignes et de colonnes de `pixels`, nombres ou tableaux de nombres prenant des valeurs entières : ces valeurs sont transférées à la carte graphique de l'ordinateur qui se charge d'afficher la couleur correspondante au bon endroit. L'essentiel du travail va donc consister à manipuler ces nombres de manière à modifier l'image de départ (ou pas).

## 24.1 Quelques remarques sur les images

Grosso-modo les fichiers image sont de deux types : ceux encodés en pixels simples, soit en général un unique entier compris entre 0 et 255 : c'est le cas des formats `pgm`, `ppm` ou `gif` qui utilisent une *palette* contenant les « vraies » couleurs, la valeur des pixels étant simplement la référence aux couleurs de la palette : exemple avec une palette de 5 couleurs…

| « valeur » du pixel | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| couleur | rouge | vert | bleu | noir | blanc |

Un fichier de ce type sera une suite de chiffres entre 0 et 4 : `01214302104`… affichera la séquence `rouge-vert-bleu-vert-blanc-noir-rouge-bleu-vert-rouge-blanc`…

Si on change la palette, on change les couleurs et on modifie donc complètement les couleurs de l'image. Cette technique permet d'avoir des images assez légères en taille bien que peu riches en couleurs mais souvent suffisantes pour Internet ; de plus on peut les regrouper facilement pour faire des petites animations[1].

Dans le deuxième type, chaque pixel est constitué d'un tableau de 3 entiers, chacun compris entre 0 et 255 (ou davantage) : chaque élément de pixel représente l'intensité d'une couleur primaire, typiquement Rouge, Verte et Bleue (système RVB, RGB en anglais) dans le système dit *additif* ou Cyan, Magenta, Jaune (système CMJ) dans le système dit *soustractif* utilisé principalement dans les travaux d'impression. Le résultat consistant dans l'ajout (ou la soustraction) de ces trois couleurs de base donne 16 millions de couleurs possibles (256*256*256). Les formats utilisés le plus couramment sont `bmp`, `jpg`, `tif`, `png`, plus ou moins intéressants en terme de qualité d'image après compression. On utilise également le format `raw` non compressé (surtout dans les appareils photo haut de gamme) permettant le travail direct sur chaque pixel.

Consulte le tableau de la page http://fr.wikipedia.org/wiki/Image_numérique pour te faire une idée plus précise.

[1] On ne parlera pas des fichiers vidéo : l'exemple des fichiers gif animés montre que sans compression les fichiers vidéo seraient beaucoup trop gros pour une utilisation normale ; si on prend une fréquence de 25 images/seconde qui est la norme du cinéma, un film de 90 mn pèserait en format 800×600 à peu près 65 Go et ne parlons pas de la 3D…

En fait en général il est intéressant d'ajouter un entier supplémentaire pour chaque pixel : la couche `alpha` en RVB (RVBA ou RGBA en anglais) qui gère la transparence de la couleur (voir http://fr.wikipedia.org/wiki/Alpha_blending).

De même on rajoute une couche Noir à CMJ qui donne l'équivalent de la transparence (CMJN ou CMYB en anglais).

Une troisième voie est utilisée avec le format vectoriel (le plus courant est le `SVG`) : au lieu d'utiliser des pixels, on stocke des primitives graphiques (droites, cercles, points, etc.) avec leurs caractéristiques (points critiques, épaisseur, couleurs, etc.) ce qui permet d'éviter les problèmes lors d'agrandissements ou réductions de taille. Évidemment c'est beaucoup plus intéressant mais ce n'est pas forcément très facile à utiliser pour les photographies !

## 24.2 Tripoter des images

Le script que nous allons étudier permet de réaliser une série d'actions sur des images du deuxième type : on peut modifier le programme pour qu'il fonctionne avec les autres formats mais ça rallongerait le script inutilement.

- Chargement d'une image.
- Affichage de l'image dans une fenêtre.
- Manipulations diverses de l'image.
- Enregistrement de l'image modifiée.

Il y a évidemment beaucoup d'autres choses réalisables mais les outils qui sont proposés forment une base de départ assez solide et il ne tient qu'à toi d'aller plus loin.

Pour manipuler les images nous allons utiliser Tkinter pour la partie interface (fenêtre, dialogues, boutons…) et la bibliothèque PIL (Public Image Library) pour le travail sur l'image. Cette bibliothèque est très complète et contient de nombreux modules utiles pour les diverses activités que tu peux envisager.

PIL est installé en standard avec Python, la documentation en anglais[1] est ici :

http://python.developpez.com/cours/pilhandbook/

ou là http://effbot.org/imagingbook/

[1] Je ne désespère pas de trouver le temps de faire une traduction… Mais bon, pas avant quelques années !

Quelques pistes en français ici : http://fsincere.free.fr/isn/python/cours_python.php

et là http://jlbicquelet.free.fr/scripts/python/pil/pil.php#information0

### 24.2.1 Les modules

```
from __future__ import division
import easydessinfr as ed
import easyguifr as eg
import Tkinter as Tk
import random as rd
import math, sys, os
from PIL import Image   as PILImage
from PIL import ImageTk as PILImageTk
```

Les classiques auxquels on ajoute deux fonctions de PIL dont les noms sont modifiés pour bien faire la différence avec les fonctions de Tkinter.

Il y a en fait plus de 20 modules différents dans PIL et sa maîtrise est loin d'être évidente.

### 24.2.2 Charger une image

Le principe est assez simple puisque c'est exactement comme pour charger un fichier « normal » avec `open`, à la différence près que PIL doit appeler le CODEC du type de fichier considéré afin d'en tirer une liste de pixels RVB.

```
def charge_fichier(nomfichier):
  imageFilename = os.path.normpath(nomfichier)
  junk,ext = os.path.splitext(imageFilename)
  if not os.path.exists(imageFilename):
    eg.boite_message('Fichier inexistant')
    fen.destroy()
  if ext.lower() in [".gif", ".pgm", ".ppm"]:
    eg.boite_message('Type de fichier pas traité')
    fen.destroy()
  return imageFilename
```

On vérifie que le nom est valide, que le fichier existe et qu'il est d'un des types utilisés ; on pourrait simplement lister les types autorisés (JPG, BMP, TIF,...).

Si ça ne va pas on détruit la fenêtre ! Inutile de tergiverser...

Maintenant on va appliquer `open` et récupérer une image en mémoire (`pil_Image`), image qui ne dépend pas du type de fichier utilisé. Une fois l'image en mémoire on crée une copie `pil_ImageAff` qui servira uniquement à l'affichage : cette copie doit rentrer dans la fenêtre quelles que soient ses dimensions, donc si elle est plus grande que la fenêtre il va falloir la réduire mais en gardant ses proportions (le **format** de l'image est égal au quotient **k_im** = **largeur/hauteur** : si on connaît la largeur on trouve la hauteur en calculant **largeur/k_im**, si on connaît la hauteur on trouve la largeur en calculant **hauteur*k_im**).

Pour l'affichage proprement dit, on utilise la fonction `Photoimage`, également disponible sous Tkinter pour les formats gif, pgm, ppm. Si tu regardes les exemples de PIL tu verras parfois l'affichage utiliser une fonction `show` qui fait appel à un gestionnaire d'image externe (en standard la visionneuse de photos sous Windows), ce qui est un peu frustrant…

Tu peux noter également quelques nouveautés comme l'utilisation (remplace `config`) de `can['width']=LA; can['height']=HA` pour modifier les dimensions du cadre ; ça marche aussi dans l'autre sens : `L=int(can['width'])` renvoie la largeur du cadre **can** dans **L** (attention, c'est une chaîne au départ, il faut la convertir en un entier).

```
def charge_image():
  global can,informe,pil_Image,im,nomfichier

  LF = int(0.8*fen.winfo_screenwidth())
  HF = int(0.8*fen.winfo_screenheight())

  imageFilename = charge_fichier(nomfichier)
  pil_Image = PILImage.open(imageFilename)
  pil_ImageAff = pil_Image.copy()

  l,h = pil_Image.size[0],pil_Image.size[1]
  k_im = l/h

  if h>HF or l>LF : HA,LA = HF,LF
  else: HA,LA = h,l
  can['width'] = LA ; can['height'] = HA
```

Les variables globales sont nécessaires pour assurer l'affichage en toutes circonstances.

**LF**, **HF** = dimensions de l'écran.

Chargement de l'image et création de sa copie d'affichage.

**l**, **h** = dimensions de l'image principale ; **k_im** = format image.

Si l'image est plus grande que la fenêtre on fixe les dimensions du cadre en fonction de la