Cours ISN Documentation

Version 20140409

Jean-Matthieu BARBER

11 June 2016

1 Table des matières 3
1.1 Architecture des ordinateurs . . . 3
1.1.1 Les composants de base d'un ordinateur . . . 3
1.1.2 Les instructions d'un processeur . . . 3
1.2 Au coeur d'un monde libre et sûr . . . 3
1.2.1 La clef ISN, le monde de GNU/Linux . . . 3
1.2.2 La ligne de commande (le shell) . . . 5
1.3 Représentation de l'information . . . 6
1.3.1 Le programme . . . 6
1.4 Réseaux . . . 19
1.5 Les bases du python . . . 19
1.5.1 Ecrire un programme . . . 19
1.5.2 Les types de base . . . 20
1.5.3 Les structures du langage . . . 24
1.5.4 Quelques exercices . . . 25
1.6 Algo . . . 26
1.7 Tutoriels . . . 26
1.7.1 Créer et programmer un micro serveur web . . . 26
1.7.2 Colorwall, un exemple de projet de jeu de plateau . . . 34

2 Index et tableaux 35

Documents de cours, mémos et exemples pour la spécialité ISN.

Avertissement : de nombreuses parties restent à compléter ou sont susceptibles de contenir des bugs/erreurs

CHAPITRE 1

Table des matières

1.1 Architecture des ordinateurs

1.1.1 Les composants de base d'un ordinateur

Démontage

- Démontage d'une unité centrale et "prise en main" des composants
  - alimentation électrique et boîtier
  - carte mère
  - processeur (et son ventilateur)
  - mémoire vive
  - disque dur
  - carte vidéo
- Comparaison avec une Raspberry pi et un téléphone.

Rôle de chaque composant

Le jeu de rôle : vis ma vie d'ordi.

Montage

- Choix et caractéristiques de chaque composant
- Constitution d'une unité centrale à prix fixé

1.1.2 Les instructions d'un processeur

Quelques exemples de programmes en assembleur.

1.2 Au coeur d'un monde libre et sûr

1.2.1 La clef ISN, le monde de GNU/Linux

Pour pouvoir travailler en mode "nomade" parmi les différentes salles info de l'établissement, au CDI ou à domicile, vous êtes dotés d'une clef ISN contenant tous les outils nécessaires aux différntes activités de l'année.

Cette clef USB contient un système d'exploitation GNU/Linux complet et peut donc être utilisée pour fournir un système d'exploitation ainsi qu'un ensemble logiciel cohérent et adapté à l'enseignement d'ISN.

Démarrage

Concrètement, il "suffit" [1] de dire à l'ordinateur de démarrer en utilisant la clef USB comme disque où chercher le système d'exploitation (selon les machines : F8, F9, Escape, Suppr, F2...).

Notes

Une fois l'ordinateur démarré sur la clef (patience, nos clefs ne sont vraiment pas des Rolls, mais c'est normalement plus rapide que le XP du lycée), vous avez accès à votre session avec le login/pass : isn/isn (vous pourrez changer votre mot de passe par la suite).

Infos utiles

Les logiciels se lancent via le petit logo Ubuntu en haut à gauche de l'écran ; il suffit de taper quelques lettres du nom du logiciel ou de ses fonctionnalités pour qu'il apparaisse.

Vous pouvez conserver un logiciel que vous utilisez fréquemment via un clic droit sur son icône lorsqu'il est lancé : "Conserver dans le lanceur".

Nous utiliserons la clef à la fois en mode "graphique" (i.e. à la souris et au clic) et en mode "rustique" (i.e. via un terminal et la ligne de commande)

Un mémo des commandes utiles (en ligne de commande) est disponible via La ligne de commande (le shell)

Exploration de l'arborescence des fichiers

Explorer (via le gestionnaire de fichier ou mieux, via un terminal en utilisant les commandes *ls* et *cd* (voir La ligne de commande (le shell)) l'architecture des répertoires de la clef.

- **/** est la racine de l'arbre (le point le plus haut - ou bas selon le sens de l'arbre)
- **/bin** contient des programmes de base du système
- **/sbin** contient des programmes de base du système, utilisables uniquement par le *superutilisateur* (sur la clef, c'est pas vous :-)
- **/lib** contient les librairies (morceaux de programme partagés entre plusieurs programmes)
- **/boot** contient les fichiers nécessaire au démarrage de l'ordinateur (le noyau, etc..)
- **/dev** est un répertoire où chaque fichier représente un "device" (composant) physique ou virtuel de la machine (exemple : /dev/sda représente le premier disque dur de la machine, /dev/sda1 représente la première partition de ce disque, etc..). Ecrire ou lire des valeurs dans ces fichiers permet d'envoyer ou de récupérer des informations de ces composants

```
$ cat /dev/random # lit sans s'arrêter des valeurs aléatoires
$ cat /dev/zero # lit des zéros
$ echo "toto" > /dev/null # envoie "toto" dans un trou noir
$ dd if=/dev/null of=/dev/sda # écrit des zéros sur tout le disque
$ ...
```

  Tous ces exemples ne sont pas exécutables par vous (surtout le dernier...)

1. Malheureusement : sur des machines récentes sous Windows 8, il peut être nécessaire de désactiver une sécurité au niveau du bios. Cette opération est en dehors des cours ISN (google is your friend : "EFI bios", à vos risques et périls - opération déconseillée)

— **/proc** est un répertoire offrant une représentation de tous les processus (programmes) en train de tourner sur la machine à chaque instant : chaque numéro contient les infos sur un programme. Ce dossier contient aussi des entrées permettant d'obtenir des informations sur certains périphériques (/proc/bus/pci/devices par exemple, ou /proc/meminfo)
— **/sys** est un répertoire présentant des fichiers permettant d'intéragir avec le système (en y lisant ou en y écrivant). Par exemple écrire la valeur adéquate dans /sys/acpi/state permet de mettre l'ordinateur en veille (http://acpi.sourceforge.net/documentation/sleep.html)
— **/etc** contient toutes les informations de configuration pour les programmes du système, ainsi que les infos sur ce qu'il faut démarrer à l'allumage du système (/etc/init..)
— **/var** contient toutes les informations qui changent régulièrement pendant que l'ordi tourne (les "logs", les mails, les fichiers à envoyer aux imprimantes, etc..)
— **/usr** reprend /bin, /sbin, /lib mais pour des programmes non vitaux pour le fonctionnement du système, ainsi que des répertoires intéressants du genre **/usr/share/doc** (toute la documentation des programmes installés sur la machine)
— **/usr/local** reprend encore cette structure (/lib, /bin, /sbin...) mais pour des programmes installés "localement" (c'est à dire manuellement, pas via le "market/store/dépôt" officiel.
— **/tmp** est le répertoire temporaire, tout le monde peut y écrire, mais les fichiers disparaissent à chaque redémarrage de la machine
— **/home** contient les répertoires utilisateurs

1.2.2 La ligne de commande (le shell)

La ligne de commande peut paraître "désuète" face aux interfaces "sexy" que l'on trouve aujourd'hui sur nos ordinateurs, cependant elle reste un outil très important pour plusieurs raisons :
— elle ne demande que peu de ressources
— elle peut être utilisée en local ou sur des machines distantes
— avec un minimum d'habitude, elle est aussi rapide que le clic de souris
— elle peut être facilement programmée pour effectuer des tâches répétitives
— elle peut permettre de sauver un ordinateur quand (presque) plus rien ne fonctionne
— etc...

L'invite de commande est la plupart du temps de la forme

```
utilisateur@machine:/chemin/du/repertoire/courant$
```

ce qui permet de savoir sur quelle machine, sous quelle identité et dans quel répertoire on travaille

Le répertoire "home" de l'utilisateur est abrégé en ~

Trucs pratiques

— **complétion automatique** : usez et abusez de la touche TAB qui complète les commandes, les noms de répertoires et de fichiers
— **copier / coller** : sélectionner un texte avec la souris, il est copié ; cliquez quelque part avec le bouton du milieu (la roulette), il est collé

Mémento des commandes utiles

Navigation et gestion des fichiers

— **cd** : change directory (changement de répertoire)
  — sans rien : retourne au répertoire home de l'utilisateur
  — avec .. (cd ..) : remonte d'un niveau dans l'arborescence des répertoires

  - avec un nom de répertoire : va dans ce répertoire
  - avec - (cd -) : retourne au répertoire dans lequel vous étiez avant de changer
- **ls** : liste des fichiers
  - sans rien : liste simple
  - avec les options -al (ls -al) : liste complète des fichiers, avec les tailles, les types, les autorisations, et incluant les fichiers cachés (ceux qui commencent par .)
- **mkdir nomrepertoire** : crée le répertoire *nomrepertoire* dans le rpertoire cournt
- **rmdir nomrepertoire** : supprime le répertoire *nomrepertoire* SI IL EST VIDE (si il n'est pas vide, il faut le vider avant)
- **rm nomfichier** : supprime le fichier *nomfichier*
- **touch nomfichier** : crée un fichier vide appelé *nomfichier*
- **man nomcommande** : affiche la page de manuel d'une commande

Réseau

- **ping xxx.xxx.xxx.xxx** : ping (icmp) une adresse ip (on peut aussi utiliser un nom de domaine à la place d'une adresse ip, dans ce cas la résolution est faite avant de lancer la commande..)
- **traceroute xxx.xxx.xxx.xxx** : renvoie la route vers une ip (ou comme ci-dessus un domaine)
- **ip addr** : affiche les interfaces réseau de la machine et leurs caractéristiques
- **whois IP/domaine** : renvoie les informations du registre sur un domaine ou une adresse IP
- **dig [@serveur] domaine** : interroge un serveur DNS (par défaut, ceux configurés par le système, sauf si @serveur est fourni) pour résolution du ddomaine (par défaut : champ A)
- **nmap IP/domaine** : scanne les ports ouverts d'une machine (moult options disponibles, RTFM)
- **wireshark** : pas une appli ligne de commande, mais qui est utile dans la catégorie "réseaux"

1.3 Représentation de l'information

1.3.1 Le programme

Dans un contexte informatique, l'information est représentée par des suites de nombres. La numérisation est l'opération qui associe à un objet réel du monde physique une description à l'aide d'un ensemble d'informations exploitables par un ordinateur ou, plus généralement, une machine numérique. À cause de l'échantillonnage sous-jacent, la numérisation induit des effets importants sur la qualité de l'information numérique. Elle entraîne des conditions spécifiques de création, de stockage, de traitement et de circulation de l'information.

Les capacités de traitement et de stockage des ordinateurs croissent de façon continue depuis leur apparition. Il est donc crucial d'organiser ces flux d'informations en local sur une machine ou de façon distribuée sur un réseau. L'intégration croissante du numérique dans les activités humaines et la numérisation de l'information suscitent des transformations culturelles, socio-économiques, juridiques et politiques profondes qui font apparaître de nouvelles opportunités, de nouveaux risques et de nouvelles contraintes qu'il convient d'étudier.

Le monde merveilleux du binaire

Dans l'ordre des activités / notions abordées :
- Compter avec 10 doigts
- Compter avec 7 doigts
- Compter avec 1 doigt
- Numération de position : décortiquage de la base 10
- Ecriture d'un nombre en binaire, conversion vers la base 10
- Ecriture d'un nombre en binaire, conversion vers la base 2
- Opérations en binaire : tables d'addition et de multiplication

— Pratique des opérations en binaire (additions, soustractions, multiplications, divisions)
— Regroupement des bits : octal, hexadécimal
— Les bits et les bytes : octet
— Les kilos, mégas, gigas, téra, pétas du binaire
— Codage d'un caractère alphanumérique : ASCII, UTF-8

Numérisation

Analogique et numérique

Signal analogique Les gandeurs physiques qui décrivent le monde qui nous entoure sont à notre échelle tout à fait continues (même si au niveau quantique, la plupart sont quantifiées), dans leurs valeurs et dans le temps.

Exemples : la température, la pression (signal sonore, par exemple), l'intensité lumineuse (vision du monde qui nous entoure, ...)

Pour transmettre une de ces informations à distance, on utilise la plupart du temps un signal électrique. Il faut alors convertir cette grandeur physique en signal électrique par l'intermédiaire d'un capteur, qui convertit la grandeur physique en grandeur électrique.

Exemples : capteur de température (thermistance, LM335, ..), capteur de lumière (photorésistance, photodiode, capteur CCD, ...), capteur de son (microphone, ...)...

Le signal électrique est alors une "image" de la grandeur physique initiale, et garde donc la continuité dans le temps et dans les valeurs : c'est un signal analogique.

On peut transmettre directement ce signal, l'encodage du signal étant donc la connaissance de la relation entre la grandeur physique initiale et le signal électrique transmis, et le décodage se faisant en inversant cette relation.

Exemple : le circuit électrique spécialisé LM335 fournit une tension électrique proportionnelle à la température (10 mV/K).

On peut aussi *numériser* ce signal.

Signal numérique La numérisation d'un signal consiste à le représenter sous la forme d'une suite de nombres (d'où le nom "numérisation").

Cette numérisation est rendue nécessaire par le fait que les ordinateurs ne sont que de grosses calculatrices qui fonctionnent avec des circuits électriques qui effectuent des opérations en binaire (0 ou 1 : voir *Bases du binaire*)

Un signal numérique est constitué d'une suite de valeurs discrètes, et donc est intrinsèquement discontinu.

L'information à décrire numériquement peut être temporelle (un son, par exemple), spatiale (une image), ou les deux à la fois (une scène animée par exemple).

Les principes de numérisation sont un peu différents selon qu'il s'agit d'une information temporelle, spatiale ou spatio-temporelle, mais certains points sont communs, comme la résolution d'un signal numérique.

Pour les détails concernant chaque type de signal, voir :
— *Numérisation d'un signal temporel*
— *Numérisation d'un signal spatial - images numériques*

Résolution d'un signal numérique Les valeurs représentées par un ordinateur sont stockées à l'aide de chiffres. Le nombre de chiffres utilisés pour décrire un élément d'information donne la résolution du signal numérique.

Exemple en base 10 :
— si on utilise 2 chiffres pour décrire une température, on a 100 valeurs disponibles. Si la plage de température que l'on décrit avec ces 100 valeurs va de 20 à 30 degrés, on aura une précision à 0,1 degrés.

- si on utilise 3 chiffres, on aura 1000 valeurs disponibles. Pour la même plage de température, on aura une précision à 0,01 degrés.
- de manière générale, avec n chiffres, on aura 10^n valeurs possibles.

Evidemment dans la nature, la température ne varie pas de 0,1 en 0,1 degrés, ni même de 0,01 en 0,01 degrés. On “manque” donc des valeurs.

Dans le monde informatique, les chiffres sont 0 ou 1, et sont appelés des **bits** ; en utilisant n bits, on peut décrire 2^n valeurs possibles.

Plus le nombre de bits utilisés est grand, meilleure est la résolution, mais cela prend aussi plus de place pour stocker l’information et plus de temps pour la transmettre. Il faut donc à chaque fois trouver le bon compromis entre la résolution et le contraintes techniques / économiques.

Exemple : pour le son qualité CD, on utilise 16 bits, soit 65536 valeurs pour décrire une “tranche” de son. Pour une photo, on utilise 3x8 bits, soit 24 bits, ce qui donne environ 16 millions de valeurs possible pour décrire un élément de photo.

Pixel : vient de l’anglais “Picture Element” (PictEl -> Pixel) : élément d’image.

Pour représenter une image sous forme numérique, on doit la découper en petit

Pixellisation

Numérisation d’un signal spatial - images numériques

La numérisation d’un signal spatial (on prendra ici l’exemple d’une image) passe d’abord par le découpage de l’image en petites zones, appelées pixels, la plupart du temps carrées (mais on peut faire des découpages rectangulaires, hexagonaux, ...)

Chaque pixel sera représenté par un nombre représentant l’intensité lumineuse.

- pour une image en noir et blanc, chaque pixel sera représenté par un nombre représentant l’intensité lumineuse moyenne de la zone du pixel.
- pour une image en couleurs, chaque pixel sera représenté par une suite de 3 nombres, représentant l’intensité de la lumière dans le domaine du rouge, du vert et du bleu (correspondant aux 3 domaines du visibles que peuvent recevoir les récepteurs situés dans notre oeil) : les “canaux” Rouge, Verts, Bleus.
- pour une image pouvant présenter une transparence, on rajoute un nombre représentant la transparence de chaque pixel : le canal “Alpha”

Le nombre de bits utilisés pour coder chaque information d’intensité lumineuse est généralement de 8 bits (soit 256 valeurs), mais on peut en utiliser moins ou plus selon les usages.

Exemples :

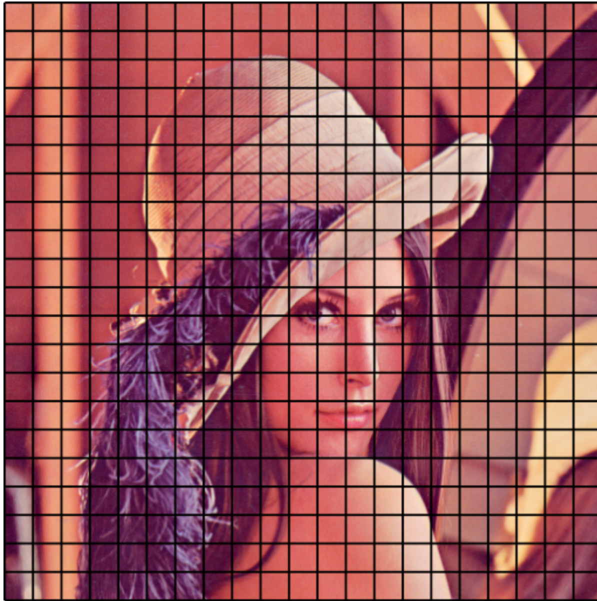
- une image strictement noire et blanc (exemple : un texte) peut être codée avec 1 bit par pixel (noir ou blanc)
- une image couleur avec transparence sera codée avec 4x8 bits.

Des exemples avec Lena

En couleurs L’image que nous prendrons pour illustrer la numérisation d’un signal spatial est l’image de Lena (miss Novembre 1972 de Playboy, devenue un standard industriel et scientifique - c’est pas une blague : http://www.cs.cmu.edu/~chuck/lennapg/

Tout d’abord voilà l’image “réelle” (c’est faux, c’est déjà une image numérisée, mais on va faire “comme si”)...

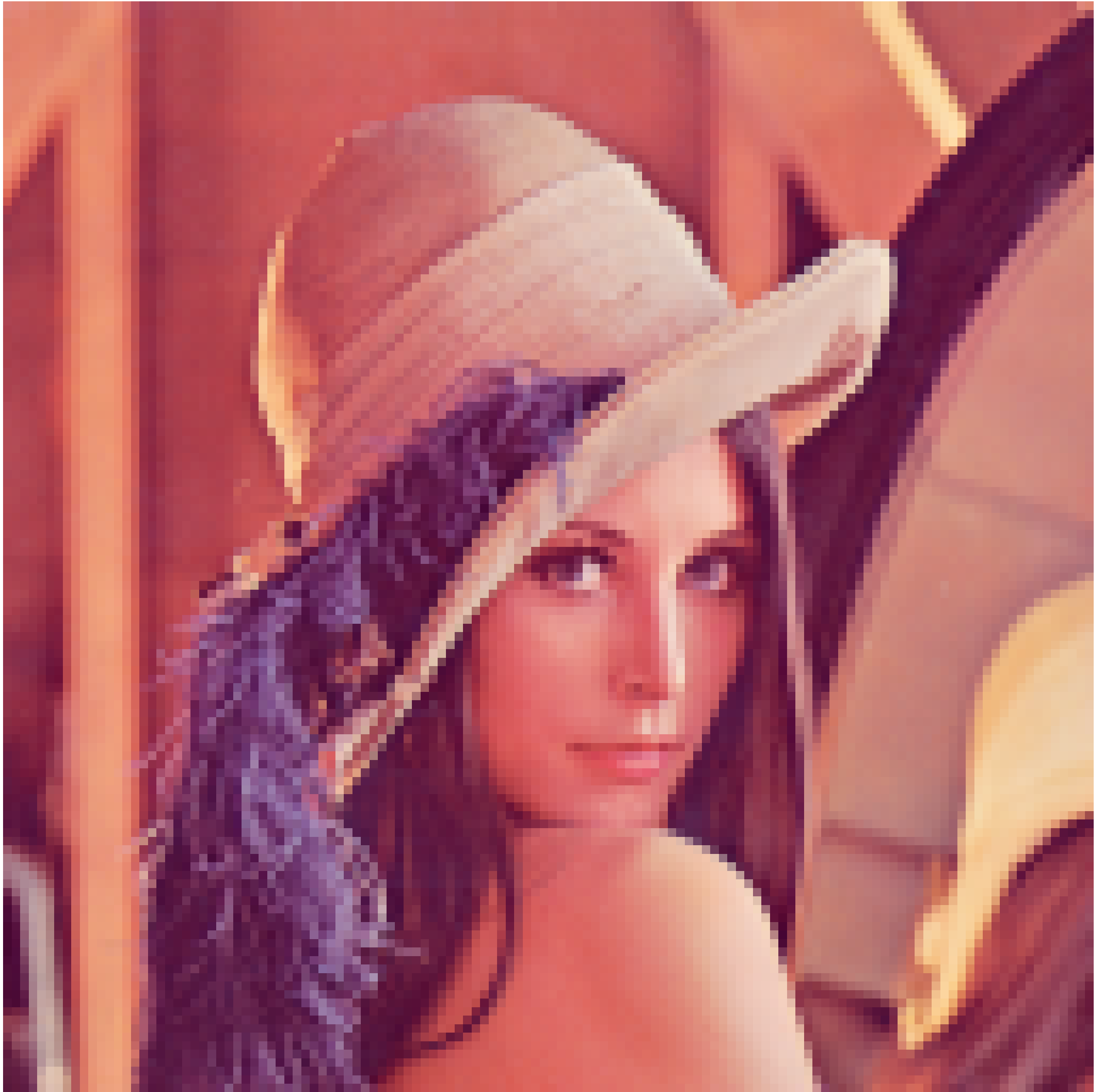
La numérisation de cette image commence par la définition d'une "grille" plus ou moins fine de pixels (PixEl = PICTure ELement à partir de l'anglais un peu déformé). Exemple : grille de 21x21 pixels :



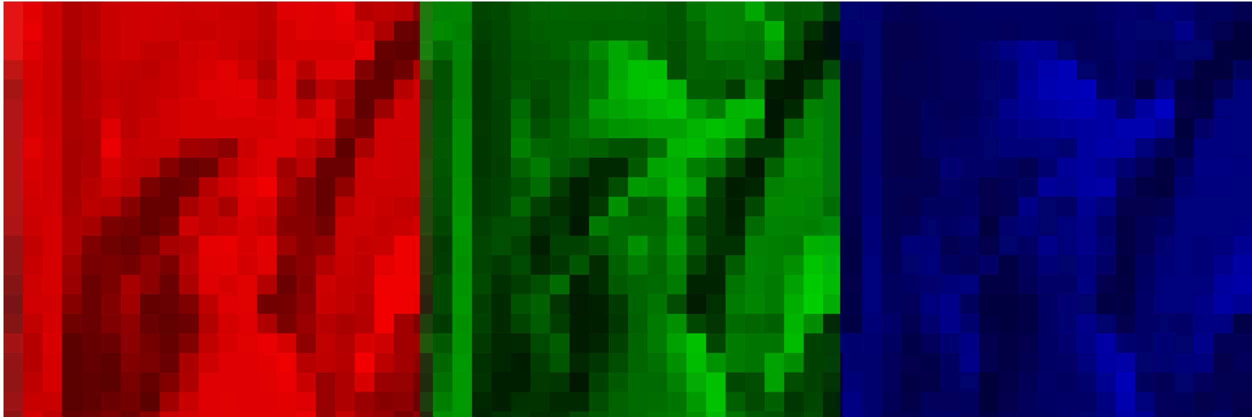
Chaque pixel est UNE entité de couleur, luminosité. Pour un appareil photo, chaque pixel est un élement photosensible du capteur de l'appareil photo (CCD). Pour un scanner, c'est la même chose, sauf que c'est une seule ligne de capteurs qui est déplacée verticalement sur l'image. Les détails en dessous du pixel sont donc "écrasés", lissés, moyennés sur toute la surface du pixel.

Lena en 21x21 pixels a donc cette tête (plus carrée et moins sympa) :

Avec 128x128 pixels, l'image serait plus reconnaissable mais ça ne change rien au principe



Puis pour chaque pixel, l'intensité de la lumière dans le rouge, le vert et le bleu sont mesurées et transformées en nombres. Le nombre de chiffres (la résolution du signal numérique) utilisés pour exprimer le résultat de cette mesure est généralement de 8 par couleur, soit 256 valeurs (0 = absence d'intensité lumineuse dans la couleur, 255 = intensité maximale dans la couleur).



- Premier pixel (en haut à gauche) : R=226, V=132, B=112
- Deuxième pixel (le suivant à droite) : R=231, V=142, B=113
- Troisième pixel : R=204, V=102, B=97
- ...
- 441ème pixel (en bas à droite) : R=122, V=46,B=72

La suite de ces valeurs forme l'image numérisée brute (raw : non compressée) : en gros : 226123112213142113.......204102097.

Dans la réalité, un ordinateur stocke des bits, donc des suites de 0 et de 1, regroupés en paquets. Il rajoute aussi souvent au début du fichier des informations sur le type de fichier, la taille du fichier, etc..

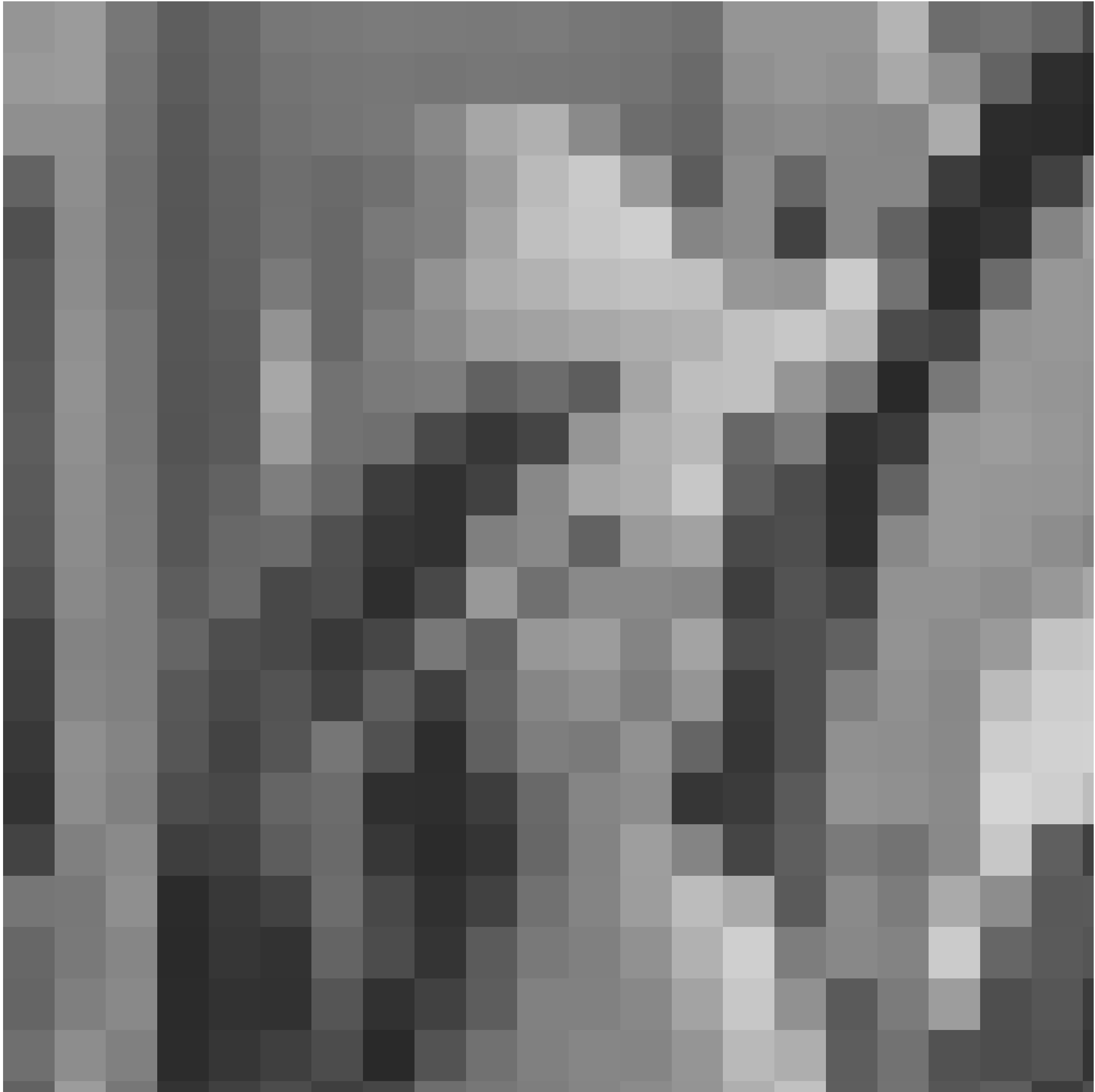
Lena en 21x21 pixels, en format de fichier TIFF (non compressé) a cette tête là

```
49 49 2A 00 34 05 00 00
E2 84 70    E7 8E 71    CC 66 61    B0 48 53    BC 52 57    CD 62 5E
D0 64 60    D1 64 61    CF 65 62    CE 63 5F    CF 65 62    CB 61 60
C2 5B 5F C6 64 64 E1 86 71 E0 81 69  E6 9B 7A D7  97 7B BC 59 5D
C4 5F 5F 9C 45 54 E5 89 70 E7 8B 6E CC 62 5A B0 45 50 BD 4F 54
CC 5E 5C D0 5F 5F D2 5F 5F CF 5D 5C C8 60 5F CB 5B 5D CE 5F 5F
C8 5B 5E BE 55 5F DF 83 73 E0 83 6D DB 7A 6A EB AD 83 C5 6A 66
96 3D 51 61 18 40 E3 80 68 E5 84 69 CC 61 5A AC 41 4E BF 4E 54
CD 5A 5A CF 5F 60 C5 63 6A C5 80 83 D2 A7 9C D6 A7 9A C8 6F 73
C2 53 5A BD 50 5A DC 77 6F DE 79 6A DB 71 63 DD 7B 69 B0 6E 6A
...
```

La première ligne est l'en-tête du fichier, on ne s'en occupe pas ici.

Les lignes suivantes représentent les pixels :
- premier pixel *E2 84 70*, en base 16 (hexadécimal, chaque chiffre représente 4 bits), correspondant à 226 (=E2), 132 (=84) et 112 (=70), valeurs R,V,B de notre premier pixel.
- deuxième pixel *E7 8E 71* correspondant à 231, 142, 113...
- etc..

En niveaux de gris Pour une image en niveaux de gris, c'est la même chose, en plus simple : il n'y a qu'une valeur à enregistrer pour chaque pixel, c'est la luminosité du point : 0 = noir, 255 = blanc.



Ce qui donne un fichier (sans l'en-tête)

```
__ __ __ __  __ __ __ __  97 9F 7C 5F  69 78 7B 7C  7B 79 7B 78  71 79 98 94  A8 A2 6E 74  58 9A 9D
5D 67 75 77  78 75 76 73  76 72 6C 95  95 8D B8 7D  52 2A 94 97  77 59 67 73  76 79 8F AF  B0 83 6B
8C 8D 87 8F  7C 24 2A 66  94 74 58 63  71 6A 71 83  A5 BF CC 86  61 8A 5F 89  82 2D 2C 66  52 93 74
63 70 69 7D  83 AC C3 C9  D5 6B 8C 2E  A1 3C 2A 47  98 5A 94 78  58 6C 6A 68  78 9A AE B2  BF C2 C1
B7 DF 32 2B  92 97 59 99  7B 56 6C 87  69 83 8E 9B  A2 A9 AB B5  C8 C9 AA 22  6C 9A 97 5D  9A 7D 57
99 71 79 7B  50 5F 45 B5  C0 B5 77 53  32 92 98 94  60 97 7E 56  5B AF 6F 68  42 34 51 B1  B2 B8 6D
26 57 A0 9C  96 5C 97 80  59 63 7F 63  32 34 44 A6  A1 B6 CA 64  32 2A 8F 98  94 93 58 96  81 58 70
...
```

soit 151, 159, 124, 95, ...

En pratique : Do It Yourself avec Gimp Les outils :

- un éditeur d'images libre : The Gimp (http://www.gimp.org/)

- un visionneur binaire : hexedit (libre) sous linux, Hexedit (gratuit, pas libre) sous windows (http://www.hexedit.com/) et iJeSaisPasMaisCaDoitExister sous mac...

TODO

En pratique : utilisation pour interférences / diffraction TODO : exploitation (RegAVI ou ImageJ) d'une photo de figure d'interférence / diffraction pour récupérer courbe intensité lumineuse.

Vocabulaire et trucs pratiques

- définition (ou taille)
- résolution
- espace disque (ou taille)

Numérisation d'un signal temporel

En pratique TODO :Audacity pour acquisition, visualisation (captures d'écran)

- échantillonnage (échantillonneur bloqueur)
- quantification
- codage

Description CAN à approximations successives, ou CAN rampe.

En TP Acquisition sons avec Audacity

- influence frequence échantillonnage (easy)
- influence résolution (8bits / 16bits) - dur à entendre, faudrait trouver moyen export vers 4bits.

Bases du binaire

Formats

Comment représenter en binaire

- un texte
- une image
- un son
- une vidéo

Compression de l'information

Chaîne de transmission d'informations

Principe général Définition : **Information** : Élément de connaissance susceptible d'être représenté à l'aide de conventions pour être conservé, traité ou communiqué.

La transmission d'une information est une activité que nous effectuons en permanence dans la vie :

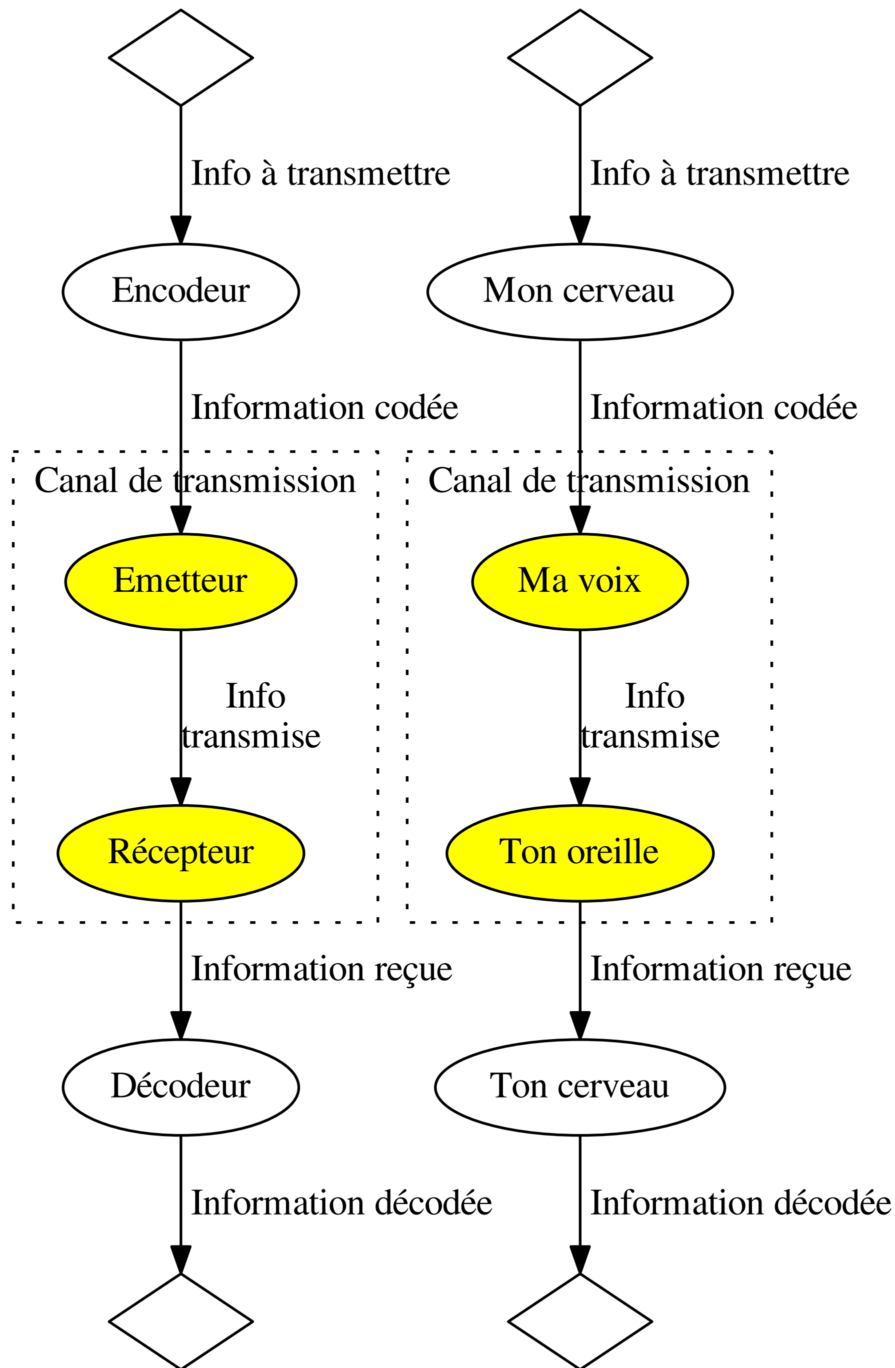
- j'ai une idée
- je l'exprime par une phrase
- je prononce la phrase
- mon interlocuteur entend la phrase
- il l'interprète et peut s'imaginer mon idée

Cette action classique est un exemple de base de transmission d'information :

- l'information à transmettre est mon idée (je suis une source d'information)

— je l'encode avec des mots, dans une langue donnée (je suis un encodeur)
— je la transmets via une onde sonore (je suis un émetteur)
— mon interlocuteur reçoit cette onde (c'est un récepteur)
— il décode le signal audio en phrase et l'interprète (c'est un décodeur)

C'est le schéma d'une chaîne de transmission d'informations :



Sur ce même schéma, on peut associer autant de chaines de transmission d'information qu'il y a de manière de coder/décoder l'information et de transmettre l'information.

Exemples : les indiens et les signaux de fumées, le télégraphe, le téléphone, la parole, le language des signes... pour

chacun de ces exemples, identifier l'encodage, l'émetteur, le canal de transmission, le récepteur et le décodeur.

Evolution des chaînes de transmission d'informations

- Avant l'électricité, le canaux de transmissions étaient visuels, sonores ou physiques (écrit..)
- L'arrivée de l'électricité a permis de rajouter un canal jusqu'alors inutilisé (invention du télégraphe)
- La maîtrise des ondes électromagnétiques a permis de s'affranchir d'un support physique pour la transmission de l'information
- Le développement de l'électronique a permis la miniaturisation des dispositifs (comparer un téléphone sans fil 1ère génération à un smartphone actuel)
- Le développement de l'informatique a permis de coder des types d'informations très variés (images, sons, vidéos, textes) et de les transmettre par des réseaux.
- Le développement de la fibre optique a permis d'augmenter le débit et la qualité des transmissions.
- L'intégration des technologies sans fil (wifi, bluetooth, téléphonie mobile) a permis de s'affranchir des liaisons filaires dans les appareils du quotidien.

Structure, organisation et persistance de l'information

- Base de données : principe base de données relationnelles (via phpmyadmin et/ou libreoffice base), principe base de données orientée documents (couchdb).
- Arborescence (exemple : yaml), parcours d'un arbre
- Suivi de version : principe, utilisation de git

1.4 Réseaux

- Le modèle OSI // TCP/IP et le système "IKEA"
- Analyse traffic réseau lycée + wifi AP open : wireshark
- Analyse trame réseau : retrouver couches, protocole
- Détail conversation http
- Routage : IP, structure réseau, traceroute, ping
- DNS, dig
- Bottle : construction mini serveur

1.5 Les bases du python

1.5.1 Ecrire un programme

Utiliser l'interpréteur python

Un programme en python peut s'écrire en mode intéractif (comme pour une calculatrice : les lignes sont exécutées au fur et à mesure qu'on les tape). On lance, dans une console, le programme `python` (ou bien `ipython` pour avoir de la couleur et quelques fonctionnalités utiles en plus) :

```
$ python
```

et on peut se servir du python comme d'une calculatrice.

```
>>> 1+1   # Addition
2
>>> 5*2   # Multiplication
10
```

Mais le mode intéractif est vite limité lorsqu'on veut écrire des programmes un peu complexes. Dans ce cas, on tape le programme en python dans un fichier texte, que l'on enregistre avec l'extension .py caractéristique des fichiers source python.

On exécute alors le programme en appelant (via la console) `python` suivi du nom du programme

```
$ python monprogramme.py
```

Attention, le programme doit se trouver dans le répertoire courant (pour vérifier quel est le répertoire courant, utiliser l'instruction `pwd` dans la console, `cd` pour changer de répertoire).

Si le programme n'est pas dans le répertoire courant, on peut l'appeler en précisant soit le chemin complet, soit le chemin relatif :

```
$ python ../../chemin/vers/le/fichier.py
$ python /home/isn/chemin/vers/le/fichier.py
```

La syntaxe d'un programme python

Tous les caractères situés derrière un # et jusqu'à la fin de la ligne sont des commentaires, ils ne sont pas exécutés.

Si le programme contient (dans les commentaires, ou dans des chaînes de caractère) au moins un caractère non ascii (accents, symboles,...), il faut débuter le programme par un commentaire "spécial" indiquant que l'encodage du programme n'est pas en ASCII. Nous utiliserons systématiquement l'UTF-8

```
# -*- coding: utf-8 -*-
```

En revanche, les noms de variables, fonctions et de manière générale toutes les "instructions" ne doivent comporter que des caractères ascii.

Les espaces sont très importants, surtout en début de ligne : c'est par l'indentation (décalage en début de ligne) que le code python est structuré. Chaque niveau de code imbriqué doit être décalé de 4 caractères par rapport au niveau supérieur.

De manière générale, la *lisibilité* du code est très importante, et permet de faire moins d'erreurs ou de repérer plus facilement les éventuelles erreurs. Un recueil des bonnes pratiques est disponibles sur le site de python : la PEP 8 (en anglais) : http://www.python.org/dev/peps/pep-0008/ écrite par le créateur du langage python, Guido van Rossum.

Les quelques points importants :
- indentation : 4 espaces (pas de tabulations)
- pas de lignes trop longues : 79 caractères maximum par ligne
- espacement des mots et symboles :
  - pas d'espace après/avant les parenthèses, accolades, crochets
  - espaces uniquement après les virgules, point-virgules ou deux points

1.5.2 Les types de base

Texte repris et traduit du tutoriel python officiel : http://docs.python.org/2/tutorial/introduction.html

Les nombres

Python effectue des opérations : `+`, `-`, `*`, `/`, `( )` fonctionnent comme sur les calculatrices, à un petit détail près : si aucun nombre ne comporte de virgule (un point, c'est de l'anglais), les calculs sont faits en nombre entier (`7/3` donne 2). En mode intéractif, cela donne :

```
>>> 2+2
4
>>> (50-5*6)/4
5
>>> 7/3
2
>>> 7/-3
-3
```

Le signe égal (=) est utilisé pour assigner une valeur à une variable. Dans ce cas, en mode intéractif, aucune valeur n'est affichée.

```
>>> largeur = 20
>>> hauteur = 5*9
>>> largeur * hauteur
900
```

Une valeur peut être assignée simultanément à plusieurs variables

```
>>> x = y = z = 0
>>> x
0
>>> y
0
>>> z
0
```

Les variables doivent être "définies" (une valeur doit leur avoir été assigné) avant de pouvoir être utilisées, sous peine de d'erreur parlant de `NameError`

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

Les nombres comportant un point sont des nombres "flottants". Si un calcul contient au moins un nombre flottant, l'ensemble du calcul est fait en flottant

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Les calculs entiers sont faits en précision "illimitée" en utilisant des entiers "longs" (terminés par L)

```
>>> 1000000000*1000000000000
1000000000000000000000L
```

On peut passer d'un format de nombre à un autre (entier / long / flottant) en utilisant les fonctions `long()`, `float()`, et `int()`

```
>>> float(1000000000000000000000L)
1e+21
>>> int(1e10)
10000000000
```

Les chaînes de caractères

Ecriture des chaines de caractères

Les chaînes de caractères peuvent s'écrire de différentes manières. Elles peuvent être entourées de guillemets simples ou doubles

```
>>> 'cours isn'
'cours isn'
>>> 'cours d\'isn'
"cours d'isn"
>>> "cours d'isn"
"cours d'isn"
>>> '"Tout à fait", qu\'il dit'
'"Tout à fait", dit-il'
>>> "\"Tout à fait\", qu'il dit"
'"Tout à fait", qu\'il dit'
```

Les chaines de caractères peuvent être écrites sur plusieurs lignes, en terminant chaque ligne par un backslash

```
hello = "Une longue chaîne de caractères qui contient : \n\
des passages à la ligne \
et des espaces."

print hello
```

Le passage à la ligne dans la chaîne se fait en utilisant le n

```
Une longue chaîne de caractères qui contient :
des passages à la ligne  et des espaces.
```

Les espaces en début de ligne sont conservés.

On peut aussi utiliser des triples-guillemets : `"""` ou `'''`

```
print """ Ceci est une
longue ligne avec des
  retours à la ligne"""
```

produit la sortie suivante

```
Ceci est une
longue ligne avec des
   retours à la ligne
```

On peut additionner des chaines de caractères (les concaténer) avec l'opérateur `+`, et les multiplier (concaténer une chaîne à elle-même un certain nombre de fois) avec l'opérateur `*`

```
>>> 'To'+'To'
'ToTo'
>>> 'To'*5
'ToToToToTo'
```

Deux chaines successives sont automatiquement concaténées

```
>>> 'To' 'To'
'ToTo'
```

Tranches de chaines (slices)

On peut récupérer une sous partie d'une chaine en utilisant un "slice" : des indices (de position), séparés par deux points.

```
>>> a="AZERTYUIOP"
>>> a[2]
'E'
```

Les positions **sont comptées à partir de zéro** pour le premier caractère. Le premier indice donné correspond au premier caractère extrait (inclus), le deuxième indice donné correspond au premier caractère non extrait (exclus).

Si un nombre n'est pas précisé, une valeur par défaut "intelligente" est choisie (longueur de la chaine, début de la chaine, fin, ..)

Les indices négatifs indiquent qu'on compte à partir de la fin (à l'envers).

```
>>> a="AZERTYUIOP"
>>> a[2:4]
'ER'
>>> a[:2]
'AZ'
>>> a[2:]
'ERTYUIOP'
>>> a[-2]
'O'
>>> a[-5:-2]
'YUI'
>>> a[:-2]
'AZERTYUI'
```

On ne peut pas modifier un caractère à l'intérieur d'une chaîne de caractères, il faut recopier la chaîne.

Pour plus de détails, voir http://docs.python.org/2/tutorial/introduction.html#strings

Listes

Le python a un certain nombre de types "composés", utilisés pour regrouper différentes valeurs. Un de ces types est la liste (list). Ecrite entre crochet, la liste a ses éléments séparés par des virgules. Comme pour les chaînes de caractères, on peut accéder à des portions de la liste en utilisant des slices.

```
>>> l=[1,2,3]
>>> l
[1, 2, 3]
>>> l[1]
2
>>> l[1:]
[2, 3]
>>> l[:2]
[1, 2]
>>> l[1::-1]
[2, 1]
>>> l[::-1]
[3, 2, 1]
>>> l[::-2]
[3, 1]
```

Mais à la différence des chaînes de caractères, on peut aussi changer un ou plusieurs éléments d'une liste.

```
>>> l[1]=9
>>> l
[1, 9, 3]
```

Un élément de liste peut être de n'importe quel type : nombre entier, flottant, chaîne de caractères, voire autre liste :

```
>>> l[2]=['A', 'zer', 'TY']
>>> l
[1, 9, ['A', 'zer', 'TY']]
```

Dans le cas d'éléments imbriqués (ci-dessus par exemple : caractère dans chaine dans liste dans liste), on peut accéder aux différents niveaux par des [] successifs.

Exemple : je veux le 3ème élément de la liste l (['A', 'zer', 'TY']), dans ce troisième élément je veux le deuxième élément ('zer'), dans de deuxième élément je veux le troisième ('r').

```
>>> l[2][1][2]
'r'
```

Voir http://docs.python.org/2/tutorial/introduction.html#lists

Dictionnaires

Les listes sont pratiques, mais le fait de ne pouvoir accéder aux éléments que par leur numéro manque un peu de "lisibilité". Pour stocker des informations qui ont un sens (exemple : un contact, avec son nom, son prénom, son mail), il est plus pratique d'utiliser un dictionnaire. Chaque élément du dictionnaire (valeur) est associé à une clef (key). A l'intérieur d'accolades, la clef (le plus souvent une chaîne de caractères), est associée à sa valeur par le symbole deux points (:). Les différentes clefs :valeur sont séparées par des virgules.

```
>>> d = {'nom': "BARBIER", 'prenom': "Jean-Matthieu", 'email':""}
>>> d
{'nom': 'BARBIER', 'prenom': 'Jean-Matthieu', 'email': ''}
```

On accède à un élément du dictionnaire en donnant la clef entre crochets :

```
>>> d['nom']
"BARBIER"
```

1.5.3 Les structures du langage

si alors / sinon-si / sinon : if elif else

Structure de if/then/else

```
if TEST1:
  instruction1
  instruction2
  ...
elif TEST2:
  instruction3
  instruction4
else:
  instruction5
  instruction6
```

TEST1, TEST2 et TEST3 sont des expressions qui sont évaluées comme des booléens (True/False). Les tests que l'on peut faire sont :

- égalité : A==B
- différence : A !=B
- supérieur : A>B (ou A>=B)
- inférieur : A<B (ou A<=B)
- négation (contraire) : not A

On peut aussi utiliser directement une valeur : les valeurs [],{},0,”” sont considérées comme False, tout le reste comme True.

Il ne faut pas oublier les : à la fin de la ligne du if/elif/else ; ce sont eux qui indiquent que la ligne de test / condition est finie, et que tout ce qui est en dessous indenté d'un cran est à exécuter si la condition est vraie. elif et else sont facultatifs.

boucle for

Exemple

```
for i in range(1,20):
  print i
```

Sortie prématurée du niveau courant de boucle : break

```
for i in range(1,20):
  if i>10:
    break
```

Retour prématuré au début de la boucle : continue

```
for i in range(1,20):
  print "."
  if i>10:
    continue
  print "-"
```

boucle while

Exemple

```
while TEST:
  instruction1
  instruction2
```

1.5.4 Quelques exercices

Exercices sur les boucles et tests

- Ecrire un programme qui affiche les 25 premiers éléments de la suite de Fibonacci (F(0)=1, F(1)=1, F(n+2)=F(n+1)+F(n))

```
    n2=0
    n1=1
    n0=0
    for i in range(2,25):
        n2=n1+n0
        n0=n1
        n1=n2
        print n2
```

— Ecrire un programme qui affiche les tables d'addition et de multiplication.
— Ecrire un programme (sans intéraction avec l'utilisateur) qui retrouve un nombre par dichotomie à l'intérieur d'un intervalle.
— Ecrire un programme qui trie de manière décroissante une liste de nombre par la méthode du tri à bulle (on parcourt la liste des nombres, si un nombre est plus petit que le suivant, on inverse leur position. Quand on arrive à la fin de la liste, on recommence - sachant qu'on peut n'aller que jusqu'au rang n-1 puisque le dernier élément de la liste est forcément le plus petit de tous)
— Ecrire un programme qui trouve tous les nombres premiers inférieurs à une certaine valeur en utilisant le crible d'Eratosthene.

Correction des exos

1.6 Algo

— algo de tri via Roberval
— tri bulle, complexité
— récursivité via factorielle
— parcours arbre via morpion

1.7 Tutoriels

Des exemples expliqués pas à pas..

1.7.1 Créer et programmer un micro serveur web

Introduction et conseils

Dans ce tutoriel, vous allez apprendre à créer et programmer un micro serveur web. Nous allons utiliser pour cela le micro framework python *bottle* (http://bottlepy.org/docs/stable/), qui a l'avantage d'être assez simple à mettre en oeuvre.

Les différentes étapes de la construction de ce projet sont enregistrées grâce au système de suivi de versions *git*, vous pourrez ainsi suivre les différentes étapes de la construction sans avoir à faire toute la saisie des lignes de code.

Lisez attentivement les instructions et ne sautez pas d'étape. A chaque pas, essayez de réfléchir à la manière dont ce que vous venez de voir pourrait vous servir pour votre projet.

Les commandes sont à taper sans erreur (ou mieux : copier/coller) dans un terminal. Une commande est toute la partie à droite de l'invite de commande (le signe $) ; ne le tapez donc pas

```
$ commande
```

Travaillez *ligne par ligne* : tapez les commandes les unes après les autres (ligne par ligne).

N'hésitez pas à utiliser un éditeur de texte (mousepad par exemple, ou scite, ou emacs...) pour regarder ce qu'il y a dans les fichiers, pour les éditer, ...

Mon premier serveur web

Etape 0 : clonage du dépôt du projet Nous allons commencer par cloner le dépôt du projet. Dans un terminal, tapez la commande suivante

```
$ git clone https://github.com/jmbarbier/ISNTutoBottle.git
```

Cette commande va cloner le dépôt ISNTutoBottle.git situé sur github vers votre machine, dans un répertoire appelé ISNTutoBottle. Ce répertoire contient tous les fichiers et tout l'historique de développement du projet.

Nous allons ensuite passer à la première étape du projet, en demandant à git de remettre le dépôt dans l'état enregistré sous le nom de step01

```
$ cd ISNTutoBottle
$ git reset --hard step01
```

La commande `git reset --hard REV/TAG` permet de remettre le répertoire dans l'état exact qui avait été enregistré lors du commit référencé REF (ou du tag TAG = référence de commit nommée par l'utilisateur).

A ce moment, si vous listez les fichiers présents dans le répertoire ISNTutoBottle (en utilisant l'explorateur de fichiers par exemple), vous ne verrez qu'un fichier "lisez-moi" (README.rst) presque vide. C'est normal, on commence tout juste.

Etape 1 : mettre en place un environnement de développement Pour pouvoir bricoler sur notre projet sans "polluer" le reste de notre système, nous allons créer un environnement python virtuel (virtualenv).

Cet environnement virtuel constitue en fait en une copie des exécutables python et des librairies utiles dans un répertoire, à partir duquel le programme python sera exécuté

```
$ virtualenv venv
```

Cette commande crèe un répertoire venv contenant tout ce qu'il faut pour exécuter des programmes en python. Deux paquets externes sont téléchargés et installés dans cet environnement virtuel : `distribute` et `pip`. Le second est le programme permettant d'installer de librairies (programmes python que vous pouvez réutiliser dans votre projet).

ATTENTION : là, votre environnement virtuel est créé (dans le répertoire venv), mais pas activé. Si vous tapez `python` dans le terminal, c'est le python de votre système qui se lancera, alors qu'on veut désormais utiliser le python de *venv*.

Pour activer un environnement virtuel, il faut taper la commande suivante

```
$ source venv/bin/activate
```

Cette commande lit (source) le fichier `activate` (activation) situé dans le répertoire `bin` du répertoire `venv` : cela active l'environnement virtuel, ce qui est visible dans l'invite de commande : avant, vous aviez une invite de commande commençant par *isn@isnusbkXX*, et maintenant, un (venv) s'est rajouté en début de ligne.

C'est le signe que l'environnement virtuel est activé (=> que lorsque vous exécutz `python`, c'est celui de l'environnement virtuel qui est appelé.

Si vous fermez votre terminal, que vous en ouvrez un deuxième, il faut refaire l'activation dans chaque terminal (vérifiez que vous avez bien le *(venv)* en début de ligne.

Etape 2 : installer les prérequis Passage à l'étape 2

```
$ git reset --hard step02
```

Dans l'explorateur de fichiers, vous devez voir apparaitre un fichier nommé *requirements.txt*.

Dans cette étape, nous allons installer les paquet / librairies / programmes externes dont nous allons avoir besoin. Le micro framework bottle consiste en un seul fichier (*bottle.py*), nous pourrions le télécharger et le coller dans le répertoire du projet, mais cette opération peut être fastidieuse si notre projet a besoin de beaucoup dépendances (librairies nécessaires), et que ces librairies doivent être mises à jour régulièrement.

On utilise donc le système `pip` qui automatise toutes ces étapes (téléchargement, installation, mise à jour, ...). `pip` comme arguments une action et un nom de librairie : `pip install bottle` par exemple va télécharger et installer la dernière version du micro framework bottle.

Et pour que n'importe qui puisse installer toutes les librairies requises pour faire fonctionner le programme, on les liste dans le fichier *requirements.txt* ; il suffit alors de lancer pip en lui disant d'installer tous les paquets requis présents dans ce fichier (un par ligne)

```
$ pip install -r requirements.txt
```

A la fin de la commande, bottle est installé dans l'environnement virtuel *venv*.

Etape 3 : créer votre premier serveur web On passe à l'étape 3

```
$ git reset --hard step03
```

Dans l'explorateur de fichiers, vous devez voir apparaître un fichier nommé `app.py`. Ce fichier est le programme principal de notre serveur web. Analysons un peu son contenu

```python
# On importe les fonctions run et route de bottle
from bottle import route, run

# On définit une route : une url à laquelle le serveur
# répondra en exécutant la fonction placée en dessous (ici
# appelée index, n'importe quel nom de fonction convient)
@route('/')
def index():
  # Cette fonction renvoie une chaîne de caractères
  # fort classique...
  return "HELLO WORLD"

# On lance le serveur, qui écoutera les requètes uniquement
# en local, sur le porte 27200, en affichant les informations
# de débuggage.
run(host='localhost', port=27200, debug=True)
```

Pour comprendre un peu mieux ce petit programme, il faut bien se rappeler les points suivants :

— la communication entre un client et un serveur se fait en TCP/IP

— le client envoie une requète au serveur (une machine identifiée par son adresse IP ou son nom de domaine), sur un port donné ; dans notre cas, la requète est une requète HTTP

— le serveur écoute le port sur l'IP, et répond à la requète.

Ici, le port d'écoute est 27200, l'IP est l'ip locale 127.0.0.1 (qui ne sort pas de la machine), et le serveur est configuré pour renvoyer le messge "HELLO WORLD" à une requète spécifique.

Nous allons tester ce serveur et décortiquer un peu son fonctionnement. Il faut d'abord le lancer, en exécutant le programme *app.py*

```
$ python app.py
```

Si tout se passe bien, le programme nous affiche quelques lignes du genre

```
(venv)isn@isnusbk01:~/ISNTutoBottle$ python app.py
Bottle v0.11.6 server starting up (using WSGIRefServer())...
Listening on http://localhost:27200/
Hit Ctrl-C to quit.
```

Traduction : je suis un serveur utilisant Bottle v0.11.6, j'écoute sur localhost, sur le port 27200 (notation IP :PORT) ; pour terminer mon exécution, appuyez sur Ctrl-C.

Prenez un navigateur, et tapez l'adresse de votre serveur dans la barre d'adresse (pas dans google ni dans le champ de recherche ! !)... votre navigateur doit afficher le message HELLO WORLD.

Et votre programme dans son terminal, a lui affiché une ou plusieurs lignes du type

```
127.0.0.1 - - [08/Mar/2013 00:00:38] "GET / HTTP/1.1" 200 11
127.0.0.1 - - [08/Mar/2013 00:00:39] "GET /favicon.ico HTTP/1.1" 404 743
```

qui sont affichées grâce au paramètre de debug, et qui précisent :

- l'adresse IP du client : ici 127.0.0.1
- la page qui fait référence à la page actuelle (lorsqu'on suit un lien sur un site, ce qui n'est pas le cas ici, le champ est donc vide : c'est l'espace entre les - -)
- la date et l'heure de la requète
- la requète HTTP
- le code de réponse
- le nombre d'octets de la réponse

Les requètes et les réponses HTTP sont composées de plusieurs parties, une lecture attentive de http://fr.wikipedia.org/wiki/Hypertext_Transfer_Protocol est fortement conseillée avant d'aller plus loin.

Ici, la **méthode** de la requète est **GET**, le client demande l'**URL /** en utilisant le protocole HTTP/1.1. Il y a aussi des en-têtes (non affichés ici), au minimum un en-tête "Host :" indiquant quel est la partie "hôte" de la demande (http://host/url).

Le serveur renvoie une réponse comprenant entre autres un code-réponse : ici 200 indique un succès, et 404 indique "page non trouvée" (le navigateur demande la favicon, l'icône pour les favoris, que notre serveur n'est pas programmé pour envoyer).

Pour mieux saisir cet échange, nous allons l'espionner un peu plus : dans votre navigateur, vous trouverez un bouton en forme de scarabée en haut à droite (FireBug). Lorsque vous l'activez, un panneau se rajoute en bas de votre navigateur, vous permettant d'explorer le code des pages que vous visitez, le trafic HTTP et beaucoup d'autres choses.

Dans le panneau de FireBug, activez l'onglet réseau (case à cocher sur la petite flèche sur l'onglet), et rechargez votre page http://localhost:27200 ... Vous voyez apparaître une ligne, indiquant qu'une requète a été faite. En développant cette ligne, vous avez toutes les informations sur la requète et la réponse, formatté sympathiquement (ou pas : vous pouvez consulter les données brutes en cliquant sur "voir le code source").

Dans le terminal, tuez votre serveur en appuyant sur Ctrl + C, et faites les exercices ci-dessous... pensez à tuer et relancer le serveur à chaque fois que vous modifiez le code source de l'application.

Exercices

- Essayez de trouver à quoi correspondent les en-têtes de l'échange HTTP que vous venez de capturer entre votre serveur et votre navigateur.
- Naviguez sur un ou deux sites avec FireBug activé, et familiarisez vous un peu avec les panneaux HTML, CSS et Réseau.
- Modifiez le programme app.py pour que votre serveur réponde "ISN" lorsque vous visitez l'adresse http://localhost:27200/ISN dans votre navigateur (en conservant le hello world pour l'url racine /)

— Cherchez dans la documentation de bottle la manière de créer une route répondant à une requète HTTP avec une méthode autre que GET (POST par exemple).

Etape 4 : utiliser des templates On passe à l'étape 4 (si le fichier app.py est encore ouvert dans un éditeur de texte, fermez le avant).

```
$ git reset --hard step04
```

Dans l'explorateur de fichiers, vous devez voir apparaître un dossier nommé `views` contenant un fichier appelé `index.tpl` :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Welcome to ISN land</title>
  </head>
  <body>
    <h1>Bienvenue {{toto}}</h1>
  </body>
</html>
```

Dans l'étape 3, nous avons renvoyé une chaîne de caractère très simple pour la route /. Pour renvoyer une page html complète, il "suffirait" de taper cette page html à la place de HELLO WORLD dans la chaîne de caractères. Mais de manière générale, dans un souci de lisibilité et d'évolutivité du code, il est déconseillé de mélanger les torchons et les serviettes : le python avec le python, le HTML avec le HTML.

Nous allons pour cela utiliser des templates (modèles) : des fichiers contenant le html à renvoyer au navigateur. Par défaut, bottle cherche ces templates dans un répertoire appelé **views**.

Si on retourne voir le fichier `app.py`, on constate les changements suivants

```
from bottle import route, run, template

#...

@route('/')
def index():
  return template('index.tpl', {'toto':'TITI'})
```

Au lieu de retourner une chaîne de caractères, on appelle la fonction `template` avec deux arguments :
— une chaîne de caractères 'index.tpl' qui désigne le template à utiliser
— un dictionnaire contenant une clef ('toto') associée à une valeur ('TITI')
La fonction *template* va aller chercher le fichier index.tpl contenu dans le dossier views, et va le renvoyer au navigateur après avoir remplacé toutes les occurrences de {{toto}} par la valeur TITI.

Si vous relancez le serveur (`python app.py`), et que vous visitez la page http://localhost:27200/ vous constaterez que s'affiche le message "Bienvenue TITI", avec un titre de page de "Welcome to ISN land"...

Il est tout à fait possible de ne pas mettre le deuxième argument de la fonction *template*, auquel cas aucun remplacement ne sera effectué dans le template.

Exercices
— Créez une route "/me" affichant une page html présentant votre nom, prénom, date de naissance.
— Créez une route "/now" affichant une page html donnant le jour et l'heure (indications : la fonction datetime.now() du module datetime permet d'obtenir l'instant présent. Pour utiliser cette fonction, il faut importer la fonction datetime du module datetime via `from datetime import datetime`.

Une fois que vous avez terminé ces premières étapes, il est désormais temps d'intéragir un peu avec notre serveur. Aller aux étapes suivantes

Intéragir avec le serveur : envoyer et traiter des informations

Etape 5 : envoyer des données au serveur

Un peu de théorie Petit rappel sur la communication entre le client et le serveur (le protocole complet est décrit dans le RFC 2616 : http://www.w3.org/Protocols/rfc2616/rfc2616.html) : le client envoie une *requète HTTP* au serveur

```
METHOD URL HTTP/1.1
Host: www.example.org
Cookie: name=value; name2=value2
HEADER: VALEUR

CONTENU DE LA REQUETE
```

- La méthode METHOD peut être GET, POST, PUT, PATCH, DELETE...
- L'URL est l'adresse que l'on envoie au serveur, de forme absolue (http : // nom.de.domaine :port chemin ? query) ou sous la forme d'un chemin (/chemin ? query). Exemples

```
http://abc.com:80/home/index.html?user=toto
http://abc.com/home/index.html
//abc.com/home/index.html
/home/index.html
```

- Les lignes suivantes sont les en-têtes http, ici *Host* qui indique (en cas d'URL uniquement sous forme de chemin par exemple) à quel nom de domaine est destinée la requète, et *Cookie* qui envoit à chaque requète les valeurs stockées dans des petits fichiers (les gâteaux) dans le navigateur du client. Il existe plusieurs dizaines d'en-têtes HTTP (http://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html#sec5.3) et on peut créer les siens (mais il vaut mieux éviter).
- Puis une ligne blanche sépare les en-têtes du contenu de la requète (si la requète a un contenu).

La transmission de données avec le serveur peut donc se faire de différentes manières selon les besoins :
- en lui envoyant une requète HTTP par la méthode **GET**
- en lui envoyant une requète HTTP par la méthode **POST**
- via les cookies (que nous traiterons + loin)
- via les en-têtes (que nous ne traiterons pas)

Il convient tout d'abord de choisir la **méthode** (GET ou POST, voire PUT, PATCH, DELETE que nous ne traiterons pas ici) à utiliser.
- GET une méthode réservée aux requètes "sûres" : en ayant en tête que les connexions internet sont souvent défectueuses, il arrive souvent que les utilisateurs s'excitent sur le bouton "envoyer" d'un formulaire, rechargent la page, etc... les requètes envoyées avec la méthode GET doivent être idempotentes, c'est à dire que même si quelqu'un recharge 20 fois la page, ça ne doit rien changer. On réserve donc le plus souvent la méthode GET aux consultations de pages n'écrivant aucune donnée du côté serveur. D'autre part, la méthode GET ne peut pas envoyer beaucoup de données puisque ces données sont envoyées via la chaîne de requète de l'url (la partie après le point d'interrogation dans http://google.fr?q=blabla). Exemple

```
GET /meteoa10jours?ville=27200
Host: www.meteofrance.com
Cookie: maville=Houlbec%20Cocherel
```

- POST (et les autres) sont des méthodes que l'on utilise pour les requètes ayant un effet sur des données côté serveur (exemple : incrémenter un compteur, ajouter une ligne dans un tableau, etc..). Les requètes POST ne sont en effet pas systématiquement renvoyées par les navigateurs (pensez au message qui s'affiche souvent

lorsque vous rechargez une page juste après avoir envoyé un formulaire). Autre avantage des requètes POST : elles peuvent contenir dans le corps même de la requète des données, sans limitation (ou peu) de taille. Exemple

```
POST /acheter
Host: www.ebay.fr
Cookie: user=myusername; code=123456

idpanier=123123&montant=3240&monnaie=EURO&article1=HomeCinema...
```

Passons à la pratique On passe à l'étape 5

```
$ git reset --hard step05
```

Dans l'explorateur de fichiers, vous devez voir apparaître dans le répertoire *views* des templates quatre fichiers appelés `form1.html`, `form2.html`, `result1.html` et `result2.html`. Quelques lignes ont aussi été ajoutées dans le fichier de l'application `app.py`.

Méthode GET `form1.html` et `result1.html` sont un exemple d'utilisation d'une méthode GET pour envoyer une donnée au serveur, et d'afficher une page de résultat. La page `form1.html` contient le formulaire :

```html
<html>
  <head>
    <title>Formulaire GET : exemple</title>
  </head>
  <body>
    <h1>Formulaire à remplir</h1>
    <form method='GET' action='/action1'>
      Votre nom : <input type='text' name='nom' /><br/>
      Votre prénom : <input type='text' name='prenom' /><br/>
      <input type='submit' value='Envoyer'/>
    </form>
  </body>
</html>
```

On peut voir ici un un formulaire, ouvert par la balise **<form>** ; les attributs de cette balise form indiquent la *méthode* (method=GET) et la page à laquelle *envoyer* le formulaire (action="/action1").

A l'intérieur de ce formulaire, on trouve du texte et des champs de formulaire :
- balise **<input type="text">** : champ de texte sur une ligne
- balise **<input type="submit">** : bouton pour l'envoi du formulaire

Il existe plusieurs types de champs de formulaire, voir w3schools.com par exemple pour plus de détails et la liste de tous leurs attributs (longueur maximale, valeur initiale, ...).

Le seul attribut obligatoire si on veut transmettre les informations rentrées dans le formulaire est l'attribut **name** : les données saisies seront associées à ce nom.

Ce formulaire est affiché lors d'une requète à l'adresse http://localhost:27200/form1 grâce aux lignes suivantes ajoutées à `app.py`

```python
# Exemple de formulaire 1 : méthode GET
@route('/form1')
def form1():
  return template("form1.tpl")

# Traitement du formulaire 1; @get permet de ne répondre qu'aux
#  requètes de méthode GET
@get('/action1')
```

```
def action1():
  # On récupère les valeurs du formulaire, envoyées dans la chaîne
  # de requète (query). La fonction get(nom, valeur_par_defaut)
  # permet de retourner une valeur par défaut si "nom" n'est pas
  # présent dans la chaîne de requète (principe de base: ne JAMAIS
  # faire confiance à l'utilisateur !!)
  nom = request.query.get('nom','INCONNU')
  prenom = request.query.get('prenom', 'INCONNU')

  # On utilise ces deux valeurs pour alimenter (via un dictionnaire)
  # l'affichage du template "result1.tpl"
  return template("result1.tpl", { 'nom': nom, 'prenom': prenom })
```

Tester : charger la page http://localhost:27200/form1 , remplir le formulaire et le soumettre. La page http://localhost:27200/action1?nom=BARBIER&prenom=Jean-Matthieu (par exemple) se charge, et on peut constater que les éléments du formulaire sont envoyés dans la chaîne de requète (la partie nom=BARBIER&prenom=Jean-Matthieu)

Méthode POST Le formulaire form2.tpl et les fonctions form2 et action2 sont quasiment identiques à form1 et action1. Les principales différences sont :

— method="POST" au lieu de method="GET" dans la balise <form>
— action="/action2" au lieu de action="/action1" dans cette même balise
— @post à la place de @get dans `app.py`
— request.forms à la place de request.query pour récupérer les valeurs transmises
— ajout des les lignes suivantes écrivent une ligne contenant le nom/prénom de l'utilisateur ainsi que la date/heure courante dans le fichier `/tmp/logins.txt`

```
    # On prend la date et l'heure courantes sous la forme
    # d'une chaîne de caractères au format ISO
    connexion = datetime.datetime.now().isoformat()

    # On sauvegarde le nom, prénom et heure de connexion dans un fichier
    with open('/tmp/logins.txt', 'a') as f:
        f.write("%s %s %s\n" % (nom, prenom, connexion))
```

Tester : charger la page http://localhost:27200/form2, remplir le formulaire et le soumettre. La page http://localhost:27200/action2 se charge et affiche un message. Vous pouvez vérifier que dans le répertoire temporaire du système (/tmp, à partir de la racine), un fichier logins.txt est apparu, et qu'il contient une ligne.

Rechargez cette page : votre navigateur devrait vous prévenir que ce n'est pas innocent. Acceptez, une deuxième ligne a dû s'ajouter dans logins.txt

Exercices

— Observer à l'aide de FireBug le dialogue entre le navigateur et le serveur, dans l'onglet "réseau". Identifier les requètes Http, les en-têtes, les query strings et le corps de la requète POST.
— Modifier le programme app.py et le fichier form1.tpl pour afficher (à la place de `result1.tpl`) le formulaire après qu'il ait été soumis, avec les valeurs des champs égales aux valeurs rentrées par l'utilisateur (indication : l'attribut **value** des champs input permet de préciser la valeur initiale).
— Modifier le programme de manière à vérifier que la longueur du nom et du prénom sont supérieures à 2. Renvoyer le formulaire avec un message d'erreur si nom ou prénom sont trop courts, et une page de succès (result1.tpl) si c'est OK.
— A partir de l'exemple donné sur le tutoriel de bottle http://bottlepy.org/docs/stable/tutorial.html#post-form-data-and-file-uploads, réaliser une page qui permet d'envoyer un fichier au serveur, ainsi que la fonction de traitement de ce formulaire, qui devra enregistrer le fichier sous un nom quelconque dans le dossier '/tmp'

1.7.2 Colorwall, un exemple de projet de jeu de plateau

Pour l'instant, sources disponibles sur http://github.com/jmbarbier/colorwall (voir les différentes étapes avec les différents commits)

CHAPITRE 2

Index et tableaux

— genindex
— modindex
— search