Introduction à l'informatique

Cours 8

kn@lmf.cnrs.fr
https://usr.lmf.cnrs.fr/~kn

université
PARIS-SACLAY

Plan

1 Présentation du cours ✓
2 Le système Unix ✓
3 Le système Unix (2) ✓
4 Python (1) : expressions, types de bases, if/else ✓
5 Python (2) : boucles, tableaux, exceptions ✓
6 Python (3) : Textes, chaînes de caractères, entrées/sorties ✓
7 Python (4) : Fonctions ✓
8 Python (5) : Concepts avancés ✓
9 Applications (1) : Introduction aux réseaux
    9.1 Principes des réseaux
    9.2 TCP/IP
    9.3 Programmation Réseau en Python

Définitions

Réseau
: ensemble de nœuds reliés entre eux par des liens (ou canaux).

Réseau informatique
: réseau où les nœuds sont des ordinateurs. Les liens sont hétérogènes (câbles, liaisons radio, liaisons satellites, ...)

Protocole
: ensemble de conventions permettant d'établir une communication mais qui ne font pas partie du sujet de la communication.

Organismes de Standardisation

Plusieurs organismes interviennent à différents niveaux

- ISO *International Organisation for Standardisation*, standardise **TOUT**. Elle publie un standard pour les réseaux informatiques (le modèle Open System Interconnection).
- IETF *Internet Engineering TaskForce*, organisme ouvert qui standardise les protocoles *TCP / IP*
- IEEE *Institute of Electrical and Electronic Engineers Standard Association*, association originellement Américaine (maintenant internationale). Definit certains standards (ex: IEEE 802.11 a/n, ...).
- W3C *World Wide Web Consortium* standardise les formats du Web comme HTML, SVG, XML, ...
- Industrie les gros acteurs ont énormément d'influence (Google, Microsoft, Cisco, ...)

Les 7 couches du modèle OSI

N	Unité	Nom	Utilisation
7	—	*Application*	Logiciel
6	—	*Presentation*	Chiffrement
5	—	*Session*	Connexion, identification
4	Segment	*Transport*	Intégrité des données
3	Paquet	*Network*	Acheminement (routage)
2	Trame	*Data-link*	Encodage sur le support physique
1	Bit	*Physical*	Matériel (voltage, nature des câbles, ...)

Chaque couche règle de manière transparente pour les couches supérieures un problème spécifique. Une couche de niveau *N* n'a aucun contrôle sur une couche de niveau inférieur.

Ce modèle constitue la norme ISO/IEC 7498-1.

(ISO 7497 le engrais, ISO 7495 concerne les farines de blé tendre, ISO 7490 les implants dentaires, ...).

Plan

1 Présentation du cours ✓
2 Le système Unix ✓
3 Le système Unix (2) ✓
4 Python (1) : expressions, types de bases, if/else ✓
5 Python (2) : boucles, tableaux, exceptions ✓
6 Python (3) : Textes, chaînes de caractères, entrées/sorties ✓
7 Python (4) : Fonctions ✓
8 Python (5) : Concepts avancés ✓
9 Applications (1) : Introduction aux réseaux
    9.1 Principes des réseaux ✓
    9.2 TCP/IP
    9.3 Programmation Réseau en Python

Modèle TCP/IP

Modèle en 4 couches, similaire au modèle OSI :

N	Nom	Description	Eq. OSI
4	*Application*	HTTP, Bitorrent, FTP, ...	5, 6, 7
3	*Transport*	TCP, UDP, SCTP, ...	4
2	*Internet*	IP (v4, v6), ICMP, IPsec, ...	3
1	*Link*	Ethernet, 802.11, ...	1, 2

Nous allons montrer comment le réseau global Internet est construit et donner des exemples de protocoles dans chaque couche.

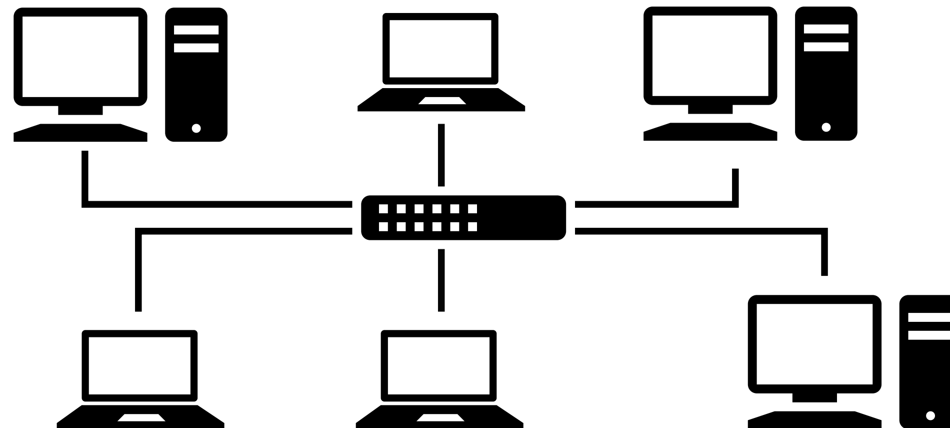
La couche de liaison

En première approximation, les protocoles de la couche de liaison sont spécifiques à un matériel particulier :

- Ethernet (cables réseaux, comme dans les salles machines)
- WiFi
- Réseaux sans fils (GSM/EDGE/3G/4G/5G)
- Réseaux cuivres (ADSL)
- Réseaux optiques (fibre)
- Réseaux sattellites
- ...

Un protocole de la couche de liaison ne permet de connecter que des machines connectées au **même type de support**.

Protocole Ethernet



- Les ordinateurs sont reliés soit directement, soit connectés à un switch.
- Chaque ordinateur a une adresse dans ce protocole, l'adresse MAC

par ex: 02:42:36:39:cc:ac

- Un ordinateur peut envoyer un paquet d'information, appelé **trame ethernet**. La trame contient (pour simplifier), l'adresse de la machine cible et les octets d'information à transmettre.
- La trame est envoyé sur le fil de cuivre (à tout le monde) et la machine de destination la lit, les autres l'ignore.

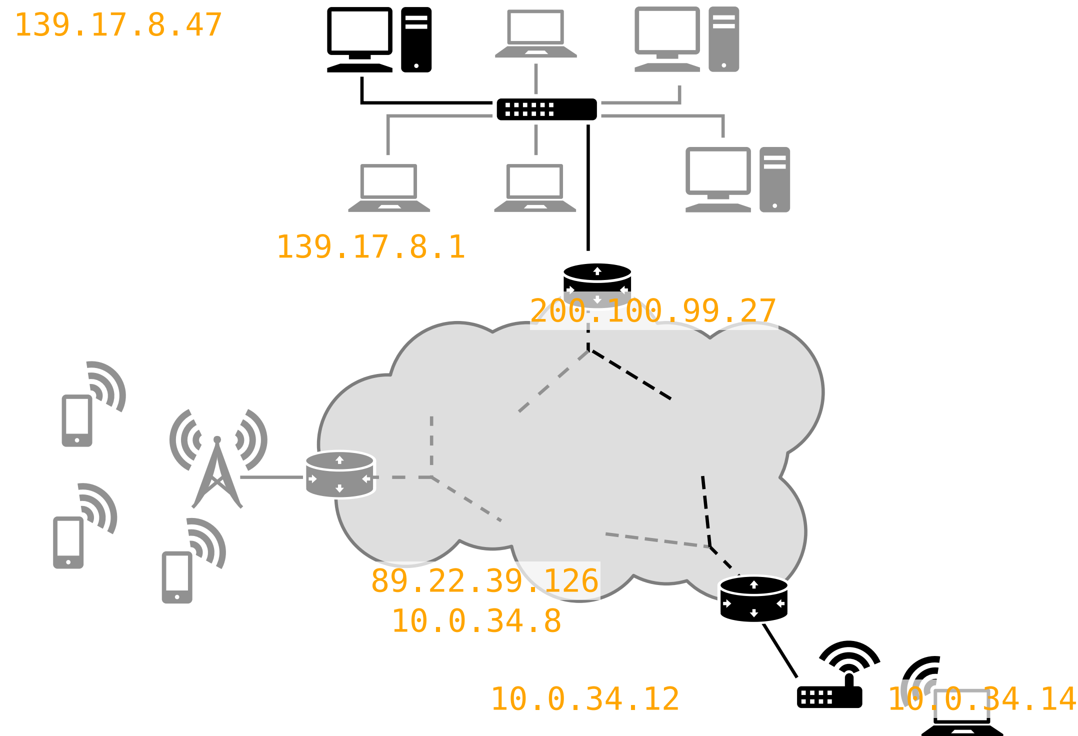
La couche Internet

Cette couche permet de connecter entre eux des **réseaux physiques différents**.

Le protocole utilisé est majoritairement IP (Internet Protocol) :

- Donne une adresse globale à chaque machine connecté
- Permet d'envoyer des paquets de données entre deux machines (même si elles ne sont pas directement reliées)
- Introduit la notion de **route**, c'est à dire les machines à traverser pour aller d'une machine A à une machine B.

Protocole IP (version 4)



- Chaque périphérique réseau est associé à une adresse de 4 octets
- Certaines machines sont à la frontières de deux réseaux physiques et ont plusieurs périphériques réseaux : *les routeurs*
- L'interconnexion de tous ces réseaux différents forment le réseau global Internet

Protocole IP (version 4)

Qui décide des adresses IP ?

- Configuration automatique (réseau local, par ex: avec une « box internet »)
- Configuration en dur : Fournisseurs d'accès, de contenu, grandes institutions, ...

Comment déterminer le chemin à prendre ?

- Si la machine de destination est sur le même réseau physique, on envoie directement.
- Sinon, on envoie les données au routeur et il se débrouille pour envoyer à un autre routeur « plus près » de la destination.
- Les données sont envoyées de proche en proche jusqu'à la destination

Adresse IP

- Moyen d'identifier une machine sur Internet
- Adresse composée de 4 octets (32 bits) :

ex: 129.175.28.179

- Certaines adresses sont réservées : 10.x.x.x, 172.[16—31].x.x, 192.168.x.x, 169.254.x.x
- L'adresse 127.0.0.1 : c'est toujours l'adresse de la machine sur laquelle on est.
- *Internet Assigned Numbers Authority*, attribue les IPs par bloc aux fournisseurs d'accès et grandes entreprises
- IPv4 est amené à disparaître, remplacé par IPv6 (adresses sur 128 bits)

Routage

Une machine A veut envoyer un message à une machine D. Les deux machines ne sont pas directement reliées (pas de câble point-à-point, pas sur le même *switch*, pas sur le même réseau Wifi, ...)

Routeur : machine possédant au moins 2 interfaces (périphériques) réseau ainsi que des logiciels spécialisés et dont le but est de transmettre les paquets IP d'un réseau vers un autre

Table de routage : spécifie, pour chaque groupe d'IPs, quel est le routeur

Exemple :

```
 $ route
Kernel IP routing table
Destination     Gateway         Genmask         ... Iface
192.168.0.0     *               255.255.255.0   ... eth1
129.175.240.0   *               255.255.255.0   ... eth0
default         129.175.240.1   0.0.0.0         ... eth0
```

Routage (suite)

Transmission d'un paquet d'une machine A1 à une machine C3

Épuisement des adresses IPv4

Combien de machines peut on adresser avec IPv4 ?

$256 \times 256 \times 256 \times 256 = 2^{32} = 4\,294\,967\,296$

Insuffisance du nombre d'adresses liée aux facteurs suivants :

- ~ 2.5 milliard d'IP uniques utilisées en 2022
- Internet of Things (montres, voitures, chaussures, ... connectées).
- Connexions permanentes (machines « toujours allumées et connectées »)
- Adressage par classe inadapté (rien entre les 16 millions d'adresses d'une classe A et les 65536 adresses d'une classe B). Si une entité veut 1 million de machines adressables, il lui faut 15 classes B.

Quelles solutions ?

IPv6 : Nouvelle version du protocole IP :

- avantage :
  - adresses sur 128 bits ($2^{128} = 3.4 \times 10^{38}$)
- inconvénient :
  - standard incompatible avec IPv4

Adressage par sous-réseau : on re-découpe la partie « machine » d'une adresse pour dénoter un sous-réseau

Réseaux locaux : certaines machines sont « cachées » du réseau global et utilisées uniquement en interne

Mise en sous-réseau

On utilise un masque *i.e.* un nombre dont le « et » binaire avec l'adresse va isoler la partie réseau.

- adresse : 129.175.34.35
- masque : 255.255.240.0 (= $11111111.11111111.11110000.00000000_2$)
- Application du masque

décimal	129	175	34	35
binaire	10000001	10101111	00100010	00100011
masque	11111111	11111111	11110000	00000000
« ET »	10000001	10101111	00100000	00000000
sous-réseau	129	175	32	0
masque inversé	00000000	00000000	00001111	11111111
« ET »	00000000	00000000	00000010	00100011
machine	0	0	2	35

Réseaux privés

Le standard réserve 3 blocs d'adresses :

- 10.0.0.0 — 10.255.255.255
- 172.16.0.0 — 172.31.255.255
- 192.168.0.0 — 192.168.255.255

Ces adresses ne peuvent jamais être données à une machine sur Internet. Elles sont utilisées par le réseau local. Seul le routeur possède une adresse publique et il s'occupe de faire la traduction entre adresses locales et adresses globales (NAT : Network Address Translation)

Comment se passe l'envoi de données ?

On veut envoyer de l'information entre une machine A et une machine B (une image, une vidéo, ...).

- La donnée est découpées en morceaux de petite taille (~ 1ko)
- Chaque morceau est annoté avec un en-tête, l'adresse IP de la source, l'adresse IP de la destination, la taille du paquet (et quelques autres méta-données) : c'est un *datagramme IP*.
- Le système d'exploitation donne le paquet IP à la carte réseau
- La carte encapsule le paquet IP dans une trame Ethernet et l'envoie à son routeur
- Le routeur (sur le réseau local Ethernet) reçoit la trame, récupère le paquet IP à l'intérieur, le réencapsule dans une autre trame physique (par exemple fibre optique) et l'envoie de proche en proche.
- La machine finale récupère le paquet IP. Elle a les données **et l'adresse IP de l'expéditeur**.

Est-ce suffisant ?

Qu'est-ce qu'il nous manque ?

- Les paquets peuvent se perdre ...
- Les paquets peuvent arriver dans le désordre
- Les paquets peuvent être corrompus ...

Autre problème : on peut avoir plusieurs applications réseaux sur la même machine (i.e. à la même adresse IP). Par exemple:

- Sur son téléphone on peut naviguer sur une page Web
- ... et écouter de la musique en streaming en même temps ...
- ... et recevoir un message via une application de messagerie ...

On ne veut pas que les données à destination des différentes applications soient mélangées.

C'est le rôle de la couche de transport.

Le protocole TCP

Le *Transmission Control Protocol* est un protocole au dessus d'IP qui permet de garantir l'intégrité des données.

Quand on veut envoyer une information d'une machine à une autre :

- La donnée est découpée en petit morceaux
- Le protocole TCP ajoute un numéro de **port source** et de **port de destination**, ainsi qu'un **numéro de séquence** : c'est un segment TCP
- Le segment TCP est encapsulé dans un datagrame IP
- Le datagrame IP est encapsulé dans une trame Ethernet

Numéro de séquence

(c'est une explication simplifiée)

Lors du découpage d'une donnée en morceaux, TCP numérote les morceaux.

- Ils sont envoyés dans l'ordre
- Lorsque la machine de destination reçoit les messages :
  - Elle peut les réordonner
  - Retirer les doublons (même numéro de séquence)
  - Redemander les morceaux manquants (numéro de séquence manquant)

Port

La couche TCP définit une notion de port. Un port est un identifiant numérique associé à une connexion TCP. Un service (ou serveur) est identifié de manière unique par son adresse IP et son numéro de port. Par convention, certains ports sont réservés pour des services particuliers :

- 21 : FTP
- 22 : SSH
- 80 : HTTP
- 110 : POP
- 443 : HTTPS

L'utilité d'un port est de pouvoir faire « tourner » plusieurs services sur la même machine (même IP).

Est-ce que c'est tout ?

On a un moyen générique et fiable d'envoyer des données de taille arbitraire entre deux machines. Une fois ce mécanisme en place, c'est le programme utilisateur qui décide de la forme des données à envoyer : c'est la couche d'application.

Exemple le protocole HTTP sur lequel on reviendra :

- Lorsque **`firefox`** se connecte à http://www.toto.com/page.html
- Il ouvre une connexion TCP à l'adresse IP correspondant au site www.toto.com, sur le port 80.
- Il envoie une chaîne de caractères :

```
GET /page.html HTTP/1.1
Host: www.toto.com:80
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/
...
```

Une application de messagerie n'enverra pas le même type de messages, et utilisera probablement un numéro de port différent.

DNS

Domain Name System : permet d'attribuer un nom à une IP (annuaire). Double avantage :

- pour les humains, un nom est plus simple à retenir
- on peut changer d'adresse IP de manière silencieuse

Principe hiérarchise :

- les serveurs DNS primaires gardent les informations sur les TLD (Top Level Domain : .com, .fr, .net, …)
- pour chaque tld, il y a un ensemble de serveur DNS de niveau 2 qui fait correspondre le nom de domaine (google.com, u-psud.fr) à un DNS de niveau 3 (généralement le DNS de niveau 2 est chez le FAI)
- le DNS de niveau 3 donne l'IP d'une machine particulière sur son domaine : mail, www (le DNS de niveau 3 est administré localement)

Plan

1 Présentation du cours ✓
2 Le système Unix ✓
3 Le système Unix (2) ✓
4 Python (1) : expressions, types de bases, if/else ✓
5 Python (2) : boucles, tableaux, exceptions ✓
6 Python (3) : Textes, chaînes de caractères, entrées/sorties ✓
7 Python (4) : Fonctions ✓
8 Python (5) : Concepts avancés ✓
9 Applications (1) : Introduction aux réseaux
- 9.1 Principes des réseaux ✓
- 9.2 TCP/IP ✓
- 9.3 Programmation Réseau en Python

Envoi de données

La programmation d'applications réseau se fait en général au niveau TCP.

◆ Serveur : le programme se met en attente de connexions sur sa carte réseau, sur un port donné. Lorsqu'une connexion arrive, elle correspond à un client qui envoie des données. Le serveur peut lire les données et éventuellement en renvoyer en échange.

◆ Client : le programme ouvre une connexion vers l'adresse IP d'un serveur sur un numéro de port. Il envoie des données et attent éventuellement une réponse du serveur.

Envoi de données en Python

On veut envoyer des données, c'est à dire des suites d'octets sur le réseau.

En Python, on ne peut **pas** utiliser des chaînes de caractères, à cause de l'encodage UTF-8.
Supposons qu'on veuille envoyer les octets : **`0x8b 0x8b = 139 139`**.
C'est une séquence UTF-8 **invalide**. Pourtant on pourrait légitimement envoyer ces octets (par exemple, parce qu'on envoie une image ou un fichier binaire, ...).

Solution : Python propose un type prédéfini de chaînes d'octets.

```
>>> b'ABCDE\x8b\x8b'
b'ABCDE\x8b\x8b'
>>> b'ABCDE\x8b\x8b' + b'toto'
b'ABCDE\x8b\x8btoto'
>>> len(b'ABCDE\x8b\x8b')
7
```

Les chaînes d'octets sont préfixées par **`b`**

Chaînes d'octets

On peut convertir une chaîne de caractères en chaîne d'octets avec l'opération **`.encode()`**. Par défaut cette opération utilise l'encodage UTF-8.

```
>>> 'ABCD'.encode()
b'ABCDE'
>>> b'ABCDÉ'.encode()
b'ABCD\xc3\x89'
>>> '🐵'.encode()
b'\xf0\x9f\x90\xb5'
```

Chaînes d'octets (2)

Les chaînes d'octets possèdent l'opération inverse **`.decode()`**

```
>>> b'ABCD'.decode()
'ABCDE'
>>> b'\xf0\x9f\x90\xb5'.decode()
'🐵'
>>> b'\x8b\x8b'.decode()
  Traceback (most recent call last):
  File "", line 1, in
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x8b in position 0: invalid
```

Attention, toutes les chaînes d'octets ne sont pas convertible en chaînes de caractères.

Le module socket

Le module **`socket`** possède un grand nombre de fonctions utilitaires :

- Connaître l'adresse IP d'une machine
- Ouvrir une connexion TCP sur un port donné vers une adresse (= client)
- Se mettre en attente de connexion TCP sur un port donné (= serveur)

gethostbyname()

La fonction **`gethostbyname(s)`** renvoie l'adresse IP de la machine dont on donne le nom.

```
>>> from socket import gethostbyname
>>> gethostbyname('www.universite-paris-saclay.fr')
'129.175.212.146'
>>> gethostbyname('www.google.fr')
'216.58.204.99'
>>> gethostnyname('www.existepas.it')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
socket.gaierror: [Errno -2] Name or service not known
```

La fonction socket

Attention, il s'agit d'une fonction **`socket`** qui est dans le module dmuême nom !

Cette fonction permet de créer une connexion réseau, par défaut en TCP.

```
>>> from socket import socket
>>> cnx = socket()
```

Une fois la connexion crée il faut choisir : soit on est un serveur, soit on est un client. On peut ensuite envoyer et recevoir des données.

`.send(…)` et `.recv()`

Une fois connecté, on peut utiliser les opérations `cnx.send(msg)` pour envoyer la chaîne d'octets `msg` à notre destinataire.

On peut aussi utiliser `cnx.recv(n)` pour lire un message de au plus `n` octets. Si on reçoit une chaîne vide, la connexion est terminée.

Attention, si on envoie et que personne ne lit, on peut saturer la connexion.
De même si on attent des octets mais que personne n'écrit, on est bloqué.

Illustration : le serveur 'ECHO'

C'est un serveur basique qui attend des connexions sur un port (on choisit 9191) et qui affiche dans la console tout ce qu'il reçoit, sans répondre.

Pour mettre une connexion en mode « serveur » on utilise l'opération `cnx.bind(a)` où `a` est une paire formée de l'adressse spéciale `'0.0.0.0'` edtu numéro de port. On peut ensuite appeler l'opération `cnx.listen()`.

Une fois la socket en mode serveur, on peut utiliser l'opération `.accept()`. Cette opération bloque le calcul et attend une connexion. Lors de la connexion, l'opération renvoie une paire : `(sclient, aclient)` formée d'une connexion vers le client spécifique qui vient de ce connecter (`sclient`) et `aclient` la paire de son adresse IP et du port sur lequel il parle.

Illustration : le serveur 'ECHO' (2)

Dans un fichier **serveur_echo.py**

```
from socket import socket

addr = ('0.0.0.0', 9191)  #adresse et port
cnx = socket()
cnx.bind(addr)            #on devient un serveur

while True:                             #pour toujours…
    sclient, aclient = cnx.listen()     #on attend un client
                                        #ici le client est connecté
    msg = sclient.recv(1024)            #on lit au plus 1000 octets
    while len (msg) != 0:               #tant que le client nous écrit
        print(">>>", msg)               #on affiche dans la console
        msg = sclient.recv(1024)        #on lit la suite
    #fin du while interne, on se remet à écouter.
```

Le client 'HELLO'

On va écrire un simple client, qui envoie le message **`HELLO`** puis se déconnecte

- Le code utilise l'opération **`cnx.connect((addr, port))`** pour dire qu'il se connecte sur l'adresse et le port où un serveur écoute
- Le client peut ensuite envoyer son message avec **`cnx.send(…)`**
- Il peut fermer explicitement la connexion avec **`cnx.close()`**

Code du client

Dans un fichier **client_hello.py**

```
from socket import socket

addr = ('127.0.0.1', 9191)
cnx = socket()
cnx.connect(addr)              #on se connecte au serveur
msg = b"HELLO"
n = cnx.send(msg)          #on envoie le message
if n != len(msg):
    print ("Le serveur n'a pas tout lu !")
cnx.close()                 #on termine
```

Conclusion

On a fait un bref survol des réseaux, c'est plutot un cours « pour la culture »

Beaucoup de détails ont été masqués. Il y a des cours de Réseaux et Réseaux avancés en L2 et L3, et un parcours de Master orienté Réseaux

Les points pricipaux

- Les réseaux reposent sur des protocoles de communication pour échanger de l'information
- Pour le réseau Internet, les protocoles sont organisés en couches, chacune ayant un rôle bien particulier :
  - Liaison : gestion du support physique
  - Internet : adressage des machines et routage des paquets
  - Transport : envoie de données sans pertes
  - Application : type de données à envoyer (application utilisateur)

On se servira des TPs pour découvrir quelques petites commandes réseaux et faire du code très simple en Python.