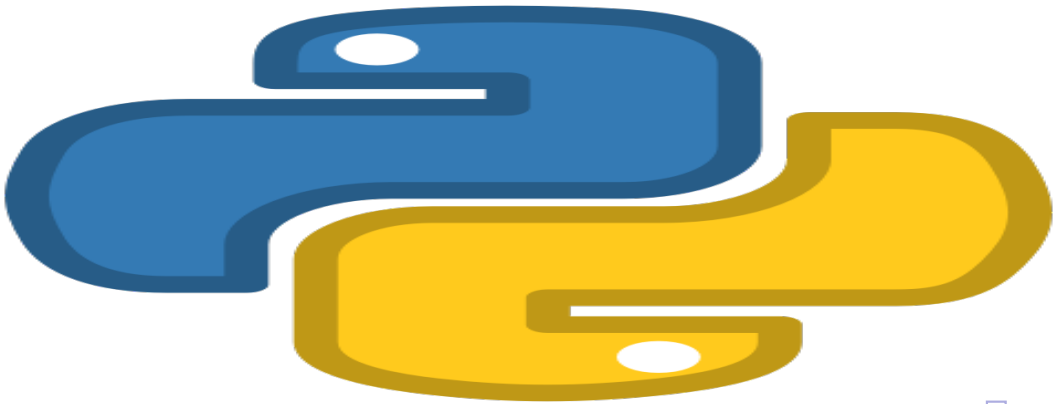
# Python : Socket

**Achref El Mouelhi**

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com



# Plan

# Plan

# Python

## Architecture client-serveur

- Un concept fondamental pour la développement des applications réparties.
- Une architecture où un serveur central interagit avec plusieurs clients.
- **Client** : Application ou utilisateur qui initie une communication en envoyant des requêtes.
- **Serveur** : Programme qui traite les requêtes des clients et retourne les réponses appropriées.

# Python

## Client : étapes

- Connexion au serveur
- **Envoi** et **réception** de données
- Fermeture de la connexion

## Serveur : étapes

- Attente d'une demande de connexion
- Acceptation de la connexion
- **Traitement** et **échange** de données
- Fermeture de la connexion

# Python

## Question

De quoi le client a-t-il besoin pour se connecter au serveur ?

# Python

## Question

De quoi le client a-t-il besoin pour se connecter au serveur ?

## Réponse

Deux informations :

- L'adresse **IP** du serveur, ou son nom d'hôte (nom du domaine),
- Le numéro de port : numéro stocké sur 16 bits (de 0 à 65 535).

# Python

## Remarques

Les numéros de port de 0 à 1023 sont réservés à des services spécifiques.

- 22 : **SSH**
- 25 : **SMTP**
- 80 : **HTTP**
- 443 : **HTTPS**
- ...

# Python

## Socket

- Module **Python**
- Permettant la communication en temps réel entre les clients et le serveur
- Fournissant une connexion bidirectionnelle persistante entre le client et le serveur
  - Mode connecté avec **TCP** (**T**ransmission **C**ontrol **P**rotocol) : une liaison est établie au préalable entre deux hôtes avant d'échanger les données.
  - Mode non-connecté avec **UDP** (**U**ser **D**atagram **P**rotocol) : les données sont échangés individuellement sans qu'une connexion soit établie.

# Python

## Socket

- Module **Python**
- Permettant la communication en temps réel entre les clients et le serveur
- Fournissant une connexion bidirectionnelle persistante entre le client et le serveur
  - Mode connecté avec **TCP** (**T**ransmission **C**ontrol **P**rotocol) : une liaison est établie au préalable entre deux hôtes avant d'échanger les données.
  - Mode non-connecté avec **UDP** (**U**ser **D**atagram **P**rotocol) : les données sont échangés individuellement sans qu'une connexion soit établie.

## Remarque

Ne pas confondre **Socket** et **WebSocket**.

# Python

## Démarche

- Créez un répertoire `python_socket` dans votre espace de travail
- Lancez **VSC** et allez dans `File > Open Folder...` et choisissez `python_socket`
- Dans `python_socket`, créez deux fichiers `serveur.py` et `client.py`.

# Python

**Remarque**

Le code présenté dans cette section sera ajouté dans le fichier `serveur.py`.

# Python

**Pour utiliser les sockets, il faut commencer par ajouter l'import suivant**

```
from socket import socket, AF_INET, SOCK_STREAM
```

# Python

**Pour utiliser les sockets, il faut commencer par ajouter l'import suivant**

```
from socket import socket, AF_INET, SOCK_STREAM
```

**Créer ensuite une instance de `socket`**

```
server_socket = socket(AF_INET, SOCK_STREAM)
```

# Python

**Pour utiliser les sockets, il faut commencer par ajouter l'import suivant**

```
from socket import socket, AF_INET, SOCK_STREAM
```

**Créer ensuite une instance de `socket`**

```
server_socket = socket(AF_INET, SOCK_STREAM)
```



## Explication

- `socket.AF_INET` : spécifie qu'il s'agit d'un socket pour une communication **IPv4** (`socket.AF_INET6` pour **IPv6**).
- `socket.SOCK_STREAM` : Indique que le socket utilise le protocole **TCP**.

# Python

**Associons la socket à une adresse et un port**

```
server_socket.bind(('0.0.0.0', 12345))
```

# Python

**Associons la socket à une adresse et un port**

```
server_socket.bind(('0.0.0.0', 12345))
```



## Explication

- `'0.0.0.0'` : signifie que le serveur écoute sur toutes les interfaces réseau disponibles de la machine.
- `12345` : port utilisé pour accepter les connexions entrantes.

# Python

**Pour mettre la socket en mode écoute**

```
server_socket.listen(5)
```

# Python

**Pour mettre la socket en mode écoute**

```
server_socket.listen(5)
```

## Explication

- 5 étant le nombre maximum de connexions en attente dans la file d'attente.
- Si le serveur reçoit plus de connexions simultanément, elles seront rejetées.

# Python

## Le code complet

```
from socket import socket, AF_INET, SOCK_STREAM

server_socket = socket(AF_INET, SOCK_STREAM)
server_socket.bind(('0.0.0.0', 12345))
server_socket.listen(5)

print("Serveur en écoute sur le port 12345")
```

# Python

**Ajoutons une boucle infinie pour permettre au serveur d'attendre afin de traiter plusieurs connexions successives**

```
while True:
    pass
```

# Python

**Pour mettre la socket en mode écoute**

```
while True:
    client_socket, client_address = server_socket.accept()
    print(f"Connexion reçue de {client_address}")
```

# Python

**Pour mettre la socket en mode écoute**

```
while True:
    client_socket, client_address = server_socket.accept()
    print(f"Connexion reçue de {client_address}")
```

## Explication

- Bloque l'exécution jusqu'à ce qu'une connexion entrante soit établie.
- Retourne deux objets :
  - `client_socket` : une nouvelle socket pour communiquer avec ce client.
  - `client_address` : l'adresse **IP** et le port du client.

# Python

**Pour fermer la connexion avec le client une fois les données échangées**

```
from socket import socket, AF_INET, SOCK_STREAM

server_socket = socket(AF_INET, SOCK_STREAM)
server_socket.bind(('0.0.0.0', 12345))
server_socket.listen(5)

print("Serveur en écoute sur le port 12345...")

while True:
    client_socket, client_address = server_socket.accept()
    print(f"Connexion reçue de {client_address}")

    client_socket.close()
```

# Python

**Remarque**

Le code présenté dans cette section sera ajouté dans le fichier `client.py`.

# Python

**Pour utiliser les sockets, il faut commencer par ajouter l'import suivant**

```
from socket import socket, AF_INET, SOCK_STREAM
```

# Python

**Pour utiliser les sockets, il faut commencer par ajouter l'import suivant**

```
from socket import socket, AF_INET, SOCK_STREAM
```

**Créer ensuite une instance de `socket`**

```
client_socket = socket(AF_INET, SOCK_STREAM)
```

# Python

**Pour utiliser les sockets, il faut commencer par ajouter l'import suivant**

```
from socket import socket, AF_INET, SOCK_STREAM
```

**Créer ensuite une instance de `socket`**

```
client_socket = socket(AF_INET, SOCK_STREAM)
```



## Explication

- `socket.AF_INET` : spécifie qu'il s'agit d'un socket pour une communication **IPv4** (`socket.AF_INET6` pour **IPv6**).
- `socket.SOCK_STREAM` : Indique que le socket utilise le protocole **TCP**.

# Python

**Associons la socket à une adresse et un port**

```
client_socket.connect(('127.0.0.1', 12345))
```

# Python

**Associons la socket à une adresse et un port**

```
client_socket.connect(('127.0.0.1', 12345))
```



## Explication

- `'127.0.0.1'` : indique l'adresse **IP** du serveur (ici, `localhost`, c'est-à-dire la machine locale).
- `12345` : port sur lequel le serveur écoute.
- Cette opération est bloquante : elle attend que la connexion soit établie avec le serveur.

# Python

**Pour fermer la connexion avec le serveur**

```
from socket import socket, AF_INET, SOCK_STREAM

client_socket = socket(AF_INET, SOCK_STREAM)
client_socket.connect(('127.0.0.1', 12345))

client_socket.close()
```

# Python

## Pour tester

- Lancer le serveur (avec `python serveur.py`) et vérifier que le message `Serveur en écoute sur le port 12345...` s'affiche.
- Lancer le client (avec `python serveur.py`) et vérifier que le message `Connexion reçue de ('127.0.0.1', 45390)` s'affiche dans le terminal du serveur.

# Python

## Pour tester

- Lancer le serveur (avec `python serveur.py`) et vérifier que le message `Serveur en écoute sur le port 12345...` s'affiche.
- Lancer le client (avec `python serveur.py`) et vérifier que le message `Connexion reçue de ('127.0.0.1', 45390)` s'affiche dans le terminal du serveur.



## Question

Pourquoi le numéro de port affiché lors de la connexion d'un client est différent de 12345 ?

# Python

## Réponse

- Le serveur est lié au port 12345. C'est le port sur lequel il écoute les connexions entrantes.
- Lorsque le client initie une connexion au serveur, le système d'exploitation du client attribue dynamiquement un port source temporaire (appelé port éphémère) pour cette connexion.
- Ainsi, en imprimant `client_address`, c'est l'adresse **IP** du client et le port source éphémère qui s'affichent.

# Python

## Explication

En **Python**, les **sockets** utilisent les méthodes suivantes pour envoyer et recevoir de données

- `send()` pour envoyer des données
- `recv()` pour recevoir des données

# Python

## Explication

En **Python**, les **sockets** utilisent les méthodes suivantes pour envoyer et recevoir de données

- `send()` pour envoyer des données
- `recv()` pour recevoir des données



## Remarque

Les données échangées, via les méthodes `send()` et `recv()`, sont de type `bytes`.
(À ne pas confondre avec `str`).

# Python

## `str` vs `bytes`

- Les `str` sont utilisés pour manipuler du texte, incluant des lettres, des chiffres, des symboles et des emojis, couvrant ainsi pratiquement tous les caractères utilisés dans les langues humaines.
- Les `bytes` sont créés en ajoutant le préfixe `b` : `b'...'` ou `b"..."`.
- Les `bytes` représentent des séquences d'octets bruts, c'est-à-dire des entiers compris entre 0 et 255, sans signification textuelle directe.
- Les `bytes` ne contiennent pas directement de caractères lisibles par l'humain. Un décodage avec l'encodage approprié est essentiel pour interpréter correctement ces octets en tant que texte.
- Les `bytes` ne permettent pas uniquement de stocker du texte sous forme binaire en utilisant un encodage approprié (comme **UTF-8**) mais des fichiers image ou des données reçues sur un réseau sous forme binaire.

# Python

## Que peut contenir le type `bytes` ?

- Le type `bytes` contient uniquement des valeurs numériques entre `0` et `255` (un octet = 8 bits).
- Ces valeurs peuvent être interprétées de deux manières :
  - **Octets imprimables (32-126)** $\Rightarrow$ lettres, chiffres, symboles visibles
  - **Octets non imprimables (0-31, 127)** $\Rightarrow$ caractères de contrôle (retour à la ligne, tabulation...)

# Python

**Pour afficher tous les caractères ASCII imprimables**

```
import string

for char in string.printable:
    print(ord(char), char)
```

# Python

**Pour afficher tous les caractères ASCII imprimables**

```
import string

for char in string.printable:
    print(ord(char), char)
```



**Pour afficher les codes ASCII, les caractères correspondants et s'ils sont imprimables**

```
for i in range(128):
    print(i, chr(i), chr(i).isprintable())
```

# Python

### Exemple avec octet imprimable

```
data = b'ABC123!@#'

print(data)
# affiche : b'ABC123!@#'
```

# Python

### Exemple avec octet imprimable

```
data = b'ABC123!@#'

print(data)
# affiche : b'ABC123!@#'
```

### Exemple avec un mélange imprimable/non imprimable

```
data = b'\x01\x02Hello\x07\x08'
print(data)
# affiche : b'\x01\x02Hello\x07\x08'
```

# Python

## Remarques

- Les lettres accentuées (é, à, ç, è...) ne font pas partie de l'**ASCII** de base (0-127).
- Elles doivent être encodées en **UTF-8** (**recommandée**), ISO-8859-1, ou un autre encodage pour être stockées dans un `bytes`.

# Python

**Pour convertir un `str` en `bytes`**

```
string = "fête"
octets = bytes(string, "utf-8")
print(octets)
# affiche b'f\xc3\xaate'
```

# Python

**Pour convertir un `str` en `bytes`**

```
string = "fête"
octets = bytes(string, "utf-8")
print(octets)
# affiche b'f\xc3\xaate'
```

**Ou**

```
string = "fête"
octets = string.encode("utf-8")
print(octets)
# affiche b'f\xc3\xaate'
```

# Python

**Pour convertir un `str` en `bytes`**

```
string = "fête"
octets = bytes(string, "utf-8")
print(octets)
# affiche b'f\xc3\xaate'
```

**Ou**

```
string = "fête"
octets = string.encode("utf-8")
print(octets)
# affiche b'f\xc3\xaate'
```

**Pour convertir un `bytes` en `str`**

```
octets = b'f\xc3\xaate'
string = octets.decode()
print(string)
# affiche fête
```

# Python

**Comme le type `str`, le type `bytes` est indexé : chaque élément est un octet**

```
data = b'\x01\x02'

print(data[0])
# affiche 1

print(data[1])
# affiche 2
```

# Python

## Question

L'accès via l'indice permet de lire un octet, mais comment lire une données écrite sur deux octets ou plus ?

# Python

## Question

L'accès via l'indice permet de lire un octet, mais comment lire une données écrite sur deux octets ou plus ?

## Réponse

Utiliser le module **STDLIB** `struct`.

# Python

**Exemple avec** `struct`

```
import struct

data = b'\x01\x02'

print(struct.unpack('>h', data))
# affiche (258,)
```

# Python

**Exemple avec `struct`**

```
import struct

data = b'\x01\x02'

print(struct.unpack('>h', data))
# affiche (258,)
```



**Pour plus de détails sur `struct`**

https://docs.python.org/fr/3.13/library/struct.html

# Python

**Pour envoyer des données du client au serveur**

```
from socket import socket, AF_INET, SOCK_STREAM

client_socket = socket(AF_INET, SOCK_STREAM)
client_socket.connect(('127.0.0.1', 12345))

client_socket.send(b"Hello serveur !")

client_socket.close()
```

# Python

**Pour envoyer des données du client au serveur**

```
from socket import socket, AF_INET, SOCK_STREAM

client_socket = socket(AF_INET, SOCK_STREAM)
client_socket.connect(('127.0.0.1', 12345))

client_socket.send(b"Hello serveur !")

client_socket.close()
```



## Explication

- Le code permet d'envoyer un message au serveur.
- Le préfixe `b` indique que la chaîne est convertie en `bytes`.

**Pour récupérer les messages du client**

```
from socket import socket, AF_INET, SOCK_STREAM

server_socket = socket(AF_INET, SOCK_STREAM)
server_socket.bind(('0.0.0.0', 12345))
server_socket.listen(5)

print("Serveur en écoute sur le port 12345...")

while True:
    client_socket, client_address = server_socket.accept()
    print(f"Connexion reçue de {client_address}")

    data = client_socket.recv(1024)
    print(f"Données reçues : {data.decode()}")

    client_socket.close()
```

**Pour récupérer les messages du client**

```
from socket import socket, AF_INET, SOCK_STREAM

server_socket = socket(AF_INET, SOCK_STREAM)
server_socket.bind(('0.0.0.0', 12345))
server_socket.listen(5)

print("Serveur en écoute sur le port 12345...")

while True:
    client_socket, client_address = server_socket.accept()
    print(f"Connexion reçue de {client_address}")

    data = client_socket.recv(1024)
    print(f"Données reçues : {data.decode()}")

    client_socket.close()
```



**Explication**

- `1024` : nombre maximum d'octets à lire.
- La méthode `recv` est également bloquante : elle attend que le client envoie des données.
- La deuxième instruction décode les données reçues (encodées en `bytes`) en chaîne de caractères et les affiche.

# Python

**Pour envoyer une message aux clients connectés**

```
from socket import socket, AF_INET, SOCK_STREAM

server_socket = socket(AF_INET, SOCK_STREAM)
server_socket.bind(('0.0.0.0', 12345))
server_socket.listen(5)

print("Serveur en écoute sur le port 12345...")

while True:
    client_socket, client_address = server_socket.accept()
    print(f"Connexion reçue de {client_address}")

    data = client_socket.recv(1024)
    print(f"Données reçues : {data.decode()}")

    client_socket.send(b"Bonjour client !")
    client_socket.close()
```

# Python

**Pour récupérer les messages du serveur**

```
from socket import socket, AF_INET, SOCK_STREAM

client_socket = socket(AF_INET, SOCK_STREAM)
client_socket.connect(('127.0.0.1', 12345))

client_socket.send(b"Hello serveur !")
response = client_socket.recv(1024)
print(f"Réponse du serveur : {response.decode()}")

client_socket.close()
```